Algorithmica (2023) 85:216–276
https://doi.org/10.1007/s00453-022-01023-w

Self-Stabilizing and Private Distributed Shared Atomic Memory in Seldomly Fair Message Passing Networks

Shlomi Dolev[1] · Thomas Petig[2] · Elad M. Schiller[2]

Received: 29 July 2018 / Accepted: 4 August 2022 / Published online: 20 August 2022

Abstract

We study the problem of privately emulating shared memory in message-passing networks. The system includes clients that store and retrieve replicated information on N servers, out of which e are *data-corrupting malicious*. When a client accesses a data-corrupting malicious server, the data field of that server response might be different from the value it originally stored. However, all other control variables in the server reply and protocol actions are according to the server algorithm. For the coded atomic storage algorithms by Cadambe et al., we present an enhancement that ensures no information leakage and data-corrupting malicious fault-tolerance. We also consider recovery after the occurrence of transient faults that violate the assumptions according to which the system was designed to operate. After their last occurrence, transient faults leave the system in an arbitrary state (while the program code stays intact). We present a self-stabilizing algorithm, which recovers after the occurrence of transient faults. This addition to Cadambe et al. considers asynchronous settings as long as no transient faults occur. The recovery from transient faults that bring the system counters (close) to their maximal values may include the use of a global reset procedure, which requires the system run to be controlled by a fair scheduler. After the recovery period, the safety properties are provided for asynchronous system runs that are not necessarily controlled by fair schedulers. Since the recovery period is bounded

An earlier version of this article appeared as a brief announcement in [26]. A technical report version of this article can be found at [30].

✉ Elad M. Schiller
elad@chalmers.se

Shlomi Dolev
dolev@cs.bgu.ac.il

Thomas Petig
petig@chalmers.se

1 Department of Computer Science, Ben-Gurion University of the Negev, 84105 Beer-Sheva, Israel

2 Department of Computer Science and Engineering, Chalmers University of Technology, 41296 Gothenburg, Sweden

and the occurrence of transient faults is extremely rare, we call this design criteria self-stabilization in the presence of seldom fairness. Our self-stabilizing algorithm uses a bounded amount of storage during asynchronous executions (that are not necessarily controlled by fair schedulers). To the best of our knowledge, we are the first to address privacy, data-corrupting malicious behavior, and self-stabilization in the context of emulating atomic shared memory in message-passing systems.

Keywords Shared memory emulation · Self-stabilization · Privacy

1 Introduction

The increasing availability of fast ubiquitous networking, the appearance of Cloud and Fog computing, have offered computer users attractive opportunities for remotely storing massive amounts of data in decentralized storage systems. In such systems, privacy and dependability are imperative. We consider distributed fault-tolerant systems that prevent information leakage, deal with *data-corrupting malicious* behavior and can recover after the occurrence of transient faults, which cause an arbitrary corruption of the system state, including the state of the mechanisms for storing information, so long as the program's code is still intact. To the best of our knowledge, we are the first to show that the emulation of atomic shared memory in message-passing systems can be done in a way that considers information privacy, resilience to data-corrupting malicious behavior and recovery from transient-faults.

1.1 The Problem

A distributed storage system uses a decentralized set of servers for allowing clients to access a shared object concurrently. Register emulation is a well-known method for sharing objects. Among the three kinds of consistency requirements for registers, atomicity is the strongest one, since it requires every sequence of concurrent access to the register to appear sequential [45]. Another classification of register emulation considers the number of clients that can read or write the shared register concurrently. We consider the more general form of shared memory emulation of an atomic register in which many clients can read and write concurrently.

1.1.1 Storage and Communication Costs

Early approaches [5, 47] provided fault-tolerance for distributed emulation of shared registers via replication. That is, each server is to store an identical copy of the most recent version of the shared object. These solutions require the read procedure to include a propagation phase in which the reader updates the servers with the most recent value they read; details appear in [47]. Since in these early approaches the interaction between the clients and the servers includes sending of the entire replica, high communication costs are implied. Recent advances in the area [15, 33] are less costly than these early approaches [5, 47], because their propagation phase messages include only the control variables, rather than the entire replica, which includes also the data field that encodes the user information whereas the control variables are just a few counters related to the replicas' bookkeeping. Moreover, using erasure coding, the servers avoid storing the entire replica by storing at the data fields only the *coded elements*, which are tailed individually to every server. This leads to further reduction in the size of messages in all phases (see further details in [44, 56]).

1.1.2 Malicious Behavior and Privacy

The use of erasure coding facilitates, as we show in this paper, the satisfaction of requirements related to data-corrupting malicious behavior and privacy. That is, when a client accesses a data-corrupting malicious server, the data field of that server response might be different from the value it originally stored (however, all other control variables in the response and protocol actions follow the algorithm). Our privacy requirement is that the collective storage of any set of less than $k_{threshold}$ servers cannot leak information, where $k_{threshold}$ is a number that we specify in Sect. 1.2.

1.1.3 Problem Description

The system has $N \in \mathbb{Z}^+$ servers that emulate an atomic shared memory, which a set of clients $M \in \mathbb{Z}^+$ may access. The coded atomic storage CAS task addresses the problem of multi-writer, multi-reader (MWMR) emulation of atomic shared memory of a single object. CAS's safety requirement says that the algorithm's external behavior follows the ones of atomic memory, and CAS's liveness requires the completion of all (non-failing) operations independently of the node availability. From the communication and storage costs, we interpret the task of CAS to restrict the messages between clients and server, as well as the storage records, to include only individualized coded elements and control variables, as in [15].

1.2 Fault Model

Our message-passing system is asynchronous and it is prone to (a) crash failures of nodes that may resume at any time in an undetectable manner, (b) packet failures, such as omission, duplication, and reordering, and (c) data-corrupting malicious servers can reply with a message that its data field is different from the originally stored value. However, all other control variables stay intact and in all other matters, data-corrupting malicious servers do not deviate from the algorithm. Thus, the studied malicious behavior that is corrupting data does not model arbitrary (Byzantine) failures. However, it fits cases in which the user data is very large and thus stored in memory segments that are more prone to (soft) errors. This is relevant, for example, for approximate memory [53], where the in-memory mechanisms for error correction are disabled in order to save energy. Another example is when Software Guard Extension (SGE) [7, 58] protects the program code and control variables, however, the large data records are not provided with the same enclave protection.

We assume that faulty nodes have an unlimited ability to coordinate their behavior. However, we assume that the data-corrupting malicious adversity cannot impersonate non-faulty servers or clients (or modify their messages). For the sake of a simple presentation, we assume non-intersection between the data-corrupting malicious server set and the set of servers that may crash.

The term quorum system, $\mathcal{Q}$, refers to all subsets of the system nodes, such that each quorum set $Q \in \mathcal{Q}$ satisfies the quorum system specifications. For example, Attiya et al. [5] specify the criterion of $\lceil \frac{N}{2} \rceil < |Q|$, Cadambe et al. [15] consider $k_{threshold} \in \{1, \ldots, N-2f\}$ and specify $\lceil \frac{N+k_{threshold}}{2} \rceil \leq |Q|$. We consider $k_{threshold} \in \{1 \ldots, N-2(f+e)\}$ and specify $\lceil \frac{N+k_{threshold}+2e}{2} \rceil \leq |Q|$ since the proposed solution requires $2e$ more servers (in each quorum intersection) than the number of servers required by Cadambe et al. We clarify that, in general, the number of faulty servers, $f+e$, must be less than $N/2$, but, as explained above, this bound is not tight for the case of the proposed solution.

We allow failing servers to resume operation at any time (from their last state while possibility losing messages during their faulty period). I.e., we do not assume detectable restarts. We assume that any failing client stops taking steps. Note that the client identifiers are recyclable using incarnation numbers (as we explain in Sect. 12.1). Thus, although M bounds the number of clients that are concurrently active, the number of client life cycles is unbounded for any practical purpose.

1.3 Self-Stabilization

In addition to the failures above, we also aim to recover after the occurrence of the last *arbitrary transient-fault* [18]. A transient-fault can model any temporary violation of assumptions according to which the system was designed to operate. This includes the corruption of control variables, e.g., the program counter and bookkeeping counters, as well as operational assumptions, e.g., $\lceil \frac{N+k+2e}{2} \rceil \leq |Q|$. Since the occurrence of these failures can be arbitrarily combined, we assume these transient-faults can alter the system state unpredictably. In particular, when modeling the system, Dijkstra [17] assumes that these violations bring the system to an arbitrary state from which a *self-stabilizing system* should recover, cf. [18]. I.e., Dijkstra requires recovery after the last occurrence of a transient-fault and once the system has recovered, it must never violate the task requirements.

Note that, in the presence of a transient-fault, an algorithm that is not self-stabilizing cannot guarantee recovery within a finite time after the occurrence of the last transient-fault. In addition, the ability to recover from transient-faults offers fault-tolerant protection complementary to the assumption above that data-corrupting malicious behavior cannot corrupt the control variable. Thus, in the event of corruption of a control variable, eventual recovery is always guaranteed.

This paper assumes the following. (i) In the absence of transient faults, the scheduler has no fairness guarantees. (ii) After the occurrence of the last transient fault, the scheduler becomes fair for a period that is at least as long as the system recovery period. Here, the scheduler refers to an adversarial entity that decides on the order in which nodes take steps as well as send and receive messages. Also, the fairness property guarantees that all non-faulty nodes can take steps as well as exchange messages infinitely often. That is, after the recovery period, the scheduler returns to offer no fairness guarantees as in (i). The proposed design criteria are called *self-stabilization in the presence of seldom fairness*. The motivation for assumption (i) is straightforward, because imposing fairness can impact the system performance since there will be a need to wait to the slowest node in the system. Note that assumption (ii) needs to be applicable rarely (and for a bounded period) since transient faults are seldom to occur. Note that such fairness assumptions can be facilitated by self-stabilizing reconfiguration of the quorum system [28]. The advantage of the proposed designed criteria is that self-stabilizing solutions for fair executions are much simpler than for ones that have no fairness guarantees.

1.4 Related Work

1.4.1 Non-self-stabilizing Register Emulation in Message-Passing Systems

The literature on (non-self-stabilizing) register emulation in message-passing systems includes single-writer multi-reader (SWMR) [5], and their multi-writer (MWMR) counterparts [33, 47], as well as solutions that provide (non-self-stabilizing) quorum reconfiguration [35, 38]. A review of related non-self-stabilizing solutions appears in [4].

[26] The literature often considers either (i) unbounded storage during asynchronous system runs that are not controlled by a fair scheduler, such as CASGC [15], AWE [2], HGR [40] and ORCAS-B [31], (ii) store, during a write operation, the entire value being written in each server, such as ORCAS-A [31], and by that incurs a worst-case storage cost, as in [5, 47], or (iii) uses a message dispersal primitive and a reliable broadcast primitive, such as [13], which during write operations, can let the storage cost to become as large as the storage cost of replication, see [15] for details. For a comprehensive review of the state-of-the-art, see [39].

In the context of self-stabilization, we cannot consider unbounded storage cost. This paper, unlike [2, 15, 31, 40], presents a bound on the storage costs also in the absence of a fair scheduler. Thus, our proposal goes beyond the state-of-the-art in the case of (i) since we consider the context of self-stabilization and privacy. Moreover, unlike [5, 13, 31, 47], during a single write operation, the added storage cost of the proposed algorithm is similar to the ones of CASGC [15].

1.4.2 Self-Stabilizing Register Emulation in Message-Passing Systems

[26] To the best of our knowledge, there is no privacy-protecting self-stabilizing solution with write operations that do not replicate the new object version among all the system servers.

Self-stabilizing emulation of shared registers exist [11, 19, 41]. But, they do not consider atomicity. Dolev et al. [20] presented a self-stabilizing algorithm for emulating atomic single-writer single-reader (SRSW) shared register in message-passing systems. This work considers many-reader and many writer (MRMW) atomic registers.

Recent solutions for shared memory emulation include practically-stabilizing emulation of SWMR registers [1], and MRMW registers [10, 29]. Pseudo-self-stabilizing emulation of atomic registers is considered in [23] for the case of SWMR.

During asynchronous system runs that are not controlled by fair schedulers, pseudo-self-stabilizing and practically-self-stabilizing systems satisfy safety requirements after an unbounded recovery period (yet finite in the former case). The case of asynchronous system runs that are controlled by fair schedulers is not considered in [23, 26] for the case of SWMR and in [10, 29] for the case of MWMR.

We do not claim that, in the presence of a fair scheduler, the solutions in [10, 23, 26, 29] have (or have not) a bounded recovery period, but we do point out that their message size is greater than the proposed algorithm by a multiplicative factor of polynomial order in the number of system nodes (in addition to the fact that their write operations replicate the new object among all servers).

Our self-stabilizing proposal has a bounded recovery period in the presence of seldomly fair schedulers. Moreover, in the absence of transient faults (that corrupt the control variables); our self-stabilizing solution works well in the absence of fair schedulers. Furthermore, one can replace the type of control variables (tags) that we use with one of the control variables in [1, 26] and abandon merely the part of our proposal that appears in Sect. 3. This replacement is straightforward. The result will be a practically-self-stabilizing variance of Cadambe et al. [15] that has a much better use of storage compared to [10, 23, 26, 29] (at a costs of polynomial factor of the message size and no bounded recovery period).

Spiegelman et al. [56] and Cadambe et al. [14] simultaneously discovered fundamental lower bounds on the storage cost of shared memory emulation that grow with the degree of concurrency. Specifically, Spiegelman et al. considered data items of D bits, concurrency degree if δ, and an upper bound on the number of storage node failures t, they show a lower bound of $\Omega(\min(t, \delta)D)$ bits on the space complexity of asynchronous distributed storage algorithms. This implies, for example, that the asymptotic storage cost can be as high as $\mathcal{O}(\delta D)$. Note that Spiegelman et al. [56] has some restrictions the coding structures that Cadambe et al. [14] does not have, yet they allow a large number of rounds. Cadambe et al. [14] considers arbitrarily flexible coding structures, but restricts the number of rounds with respect to Spiegelman et al. [56]. Our upper bound on the storage size (Sect. 11) does not contradict the lower bound in [14, 56] and Spiegelman et al.'s $\Theta(\min(t, \delta)D)$ upper-bound does not consider self-stabilization.

To the best of our knowledge, the literature in the area of coded atomic storage [15, 16, 42–44] does not consider self-stabilization. In particular, ARES [16] supports reconfigurable-shared atomic memory emulation that uses erasure coding with only two rounds of message exchanges for a client operation. See [49] for a survey on (non-self-stabilizing) reconfigurable solutions to memory emulation. We note the existence of a self-stabilizing service for quorum reconfiguration [28] that can be combined with the proposed solution as a possible extension.

1.4.3 Privacy-Preservation

The CAS algorithm [15] uses erasure codes for splitting the data into different coded elements that each server stores. As long as at least $k_{threshold}$ coded-elements are available, the algorithm can retrieve the original information. Cadambe et al. [15] show how to use $(N, k_{threshold})$-maximum distance separable (MDS) codes [51] for improving communication and storage performances. $(N, k_{threshold})$-MDS codes map $k_{threshold}$-length vectors to an N-length ones. The CAS algorithm lets the writers to store on N servers $k_{threshold}$-length vectors. Each of the N servers stores (uniquely) one of the N coordinates of the $(N, k_{threshold})$-MDS-coded information. When retrieving the information, the algorithm can tolerate up to $(N - k_{threshold})$ erasures. We use a variation on the CAS algorithm [15] that uses Reed-Solomon codes [48] for facilitating a privacy-protection solution by storing on each server merely parts of the data, as in Shamir's secret sharing scheme [54], which we can implement by Reed-Solomon codes [48] and a matching error correction algorithm (Berlekamp-Welch [57]).

From the point of view of privacy-protection, this work focuses on the content of the records that users store in the system. We consider the case that a curious entity has the power to compromise $k_{threshold} - 1$ servers, i.e., retrieve any information that has ever arrived to them, where $k_{threshold}$ is a known constant. Shamir's secret sharing scheme guarantees that the ability to compromise $k_{threshold} - 1$ servers does not allow the curious entity to efficiently know anything about the content of the user records.

1.4.4 Proposed Techniques of Independent Interest

Note that our proposal enhances CASGC [15] from the privacy perspective, as well as from the system robustness point-of-view. To that end, we use several techniques of independent interest that facilitate this improvement.

Such techniques are needed, for example, when our solution deals with the following interesting challenge. The rate in which clients complete write operations can be much faster than the rate in which these clients can inform all the servers about these operations. This rate can also exceed the rate in which the servers can inform each other about such updates. The challenge here is imposed by the fact that self-stabilizing end-to-end protocols must assume that the communication channels have bounded capacities due to well-known impossibility results [18, Chapter 3.2]. Our solution overcomes this challenge using techniques that resemble the ones for converting shared memory models to message-passing ones [18, Chapter 4.2] and an extra phase in the writer procedure. This part of the solution is another key difference between the proposed algorithm and the one by Cadambe et al. [15].

To the end of bounding the number records that each server needs to store, at any point of time, a given server record is considered to be relevant only as long as the servers use it. We show that no server store more than $N + \delta + 3$ relevant records during asynchronous system runs that are not necessarily controlled by a fair scheduler, where δ is a bound on the number of write operations that occur concurrently with any read operation; this is similar to the δ parameter defined by Cadambe et al. [15]. The proof technique serves as a self-stabilizing alternative to existing non-self-stabilizing algorithms that provide bounds on the number of records at the server storage, such as [13, 31], in a way that does not require storage costs during write operations to be the ones of a fully replicated solution.

Georgiou et al. [37] have implemented the algorithm proposed by Cadambe et al. [15] and the one proposed here. Their experiments on PlanetLab show that our algorithm scales very well with respect to the number of servers, the number of concurrent clients, and the replicated object size. Furthermore, they claim that our algorithm has only a constant overhead compared to the algorithm proposed by Cadambe et al. [15]. They also claim that their experiments demonstrated a rapid recovery period that is proportional to just a few client operations. We note that Georgiou et al. [37] provide a validation of the analysis presented here.

1.5 Our Contributions

We present the algorithmic design for an important component for dependable distributed systems: a robust shared storage that preserves privacy. In particular, we provide a privacy-preserving and self-stabilizing algorithm for decentralized shared memory emulation (over asynchronous message-passing systems) that is resilient to a wide spectrum of node and communication failures, as well as data-corrupting malicious behavior. Moreover, our self-stabilizing algorithm can automatically recover after the occurrence of transient faults that violate the assumptions according to which the system is to behave. Concretely, we present, to the best of our knowledge, the first solution that provides:

1. *Dependable and efficient emulation of atomic registers over asynchronous message-passing systems.* When starting from a legitimate state, our self-stabilizing solution can:
   - *Deal with communication failures* The communication channels that are prone to packet failures, such as omission, duplication, reordering, but the resulted communication delays are unbounded, yet finite since we assume communication fairness. (That is, it might take a finite number of retransmissions, but packets are received eventually.)
   - *Deal with node failures* We show that non-failing clients can retrieve information stored privately by the $N - f$ non-failing servers. We do not bound the number of failing clients but we do assume a bound of M on the number of concurrently active clients.
   - *Deal with data-corrupting malicious behavior* We show that the client can retrieve the originally stored object in the presence of at most e data-corrupting malicious servers.
   - *Prevent information leakage* We show that the collective storage of any set of fewer than $k_{threshold} - 1$ servers cannot reveal (any version) of the object.
2. *Recovering after the occurrence of transient failures* We show that our algorithm can recover even after the occurrence of transient failures in the following cases. The solution presentation considers two 'attempts' to solve the problems until the third attempt provides a self-stabilizing solution.

- *Unbounded control variables and number records at the server storage.* We show that starting from an arbitrary system state and within $\mathcal{O}(1)$ time of fair execution, the system reaches a legitimate state after which the algorithm satisfies the CAS's task requirements even when the scheduler stops being fair and the execution becomes asynchronous. This 'first attempt' solution assumes that the servers can store all the object versions (in addition, stale information originated from the system starting state).
- *Unbounded control variables but a bounded number of records at the server storage:* We bound the number of relevant records that any server stores, at any point of time, by $N + \delta + 3$ during asynchronous system runs that are not necessarily controlled by a fair scheduler, where δ, similar to Cadambe et al. [15], is a bound on the number of write operations that occur concurrently with any read operation.
- *Bounded control variables and number of records at the server storage* The challenge here comes from the fact that any transient fault can bring the control variables to their maximal values. The difficulty here is that there is a need to allow the system to perform an unbounded number of write operations after this overflow event. We address this challenge by using a safety-preserving global restart of the control variables (in a way that may temporarily violate liveness but will leave the most recent version of the object intact).

Another important contribution of this work is the proposal of a new design criterion for self-stabilizing systems, that of self-stabilization in the presence of seldom fairness. On the one hand, the proposed design criteria consider a greater set of algorithms that can be considered self-stabilizing when compared to other design criteria [1, 12, 23, 29, 52] that do not consider execution fairness at all, not even seldomly. On the other hand, it is much easier to design algorithms for the proposed design criteria than the ones in [1, 12, 23, 29, 52].

1.6 Solution Outline and Document Organization

We bring our interpretation of the system in the self-stabilization context and the CAS task (Sect. 2.1) before bringing Cadambe et al. [15] version of CAS (Sect. 3). We present our privacy-preserving variation of Cadambe et al.'s algorithm as a basic result (Sect. 4).

Our self-stabilizing algorithm requires the specification of a formal model (Sect. 2.2) and external building blocks (Sect. 5). The presentation of this algorithm starts by considering its unbounded version (Sect. 7) together with its correctness proof (Sect. 8). Our proof also shows that there is a bounded set of relevant records that the servers store (Sect. 9). This bound is the basis for the bounded variation of the proposed self-stabilizing algorithm (Sect. 10) and its cost analysis (Sect. 11).

The transformation of the unbounded solution into its bounded variation is facilitated by a self-stabilizing global reset that requires the participation of all nodes. Thus, the self-stabilizing and bounded part of our contribution fits systems in which nodes can host three different kinds of services: server, client, and reset. For the sake of a simple presentation, we assume each node hosts exactly one instance of each of the three services and that the total number of nodes is simply bounded by N and that the number of clients and servers is the same, i.e., $N = M$. We clarify that, at any time, each server could be accessed *concurrently* by any subset of the N client services hosted by the nodes. In addition, an extension of the proposed solution in which nodes host any predefined number server services and any predefined number of client services is straightforward.

The discussion (Sect. 12) includes also an elegant extension that extends our settings to consider the possible recovery of failing nodes. We present self-stabilizing implementations (Sect. 6) of the gossip and quorum services (specified in Sect. 5). We note that the use of gossip is borrowed from the studied algorithm by Cadambe et al. Also, as Georgiou et al. [37] demonstrated empirically, there is a straightforward trade-off between the rate in which the gossip messages are sent and the time it takes the system to recover after the occurrence of the last transient-fault. Thus, one can easily avoid saturating the network bandwidth by using a low gossip rate.

2 System Settings

This section describes the system and brings its related assumptions. Section 2.1 overviews the assumptions needed for understanding the solution by Cadambe et al. [15] (which we review in Sect. 3) and our own (non-self-stabilizing) contribution presented in Sect. 4. Section 2.2 presents the assumptions needed for our self-stabilizing solutions and proofs presented in Sects. 5 to 11.

2.1 System Overview

The design criteria of self-stabilization have considerations that must be taken into account (in addition to the ones that exist for non-stabilizing systems). Therefore, before describing the algorithm by Cadambe et al. [15] and proposing our variation (Sects. 2.2 to 11), this section brings the studied task (Sect. 2.1.1) and our interpretation of the system contexts that do (Fig. 1, right), and do not (Fig. 1, left) consider privacy and self-stabilization. We note that in the context of self-stabilization, all system components have to follow the self-stabilization criteria. Thus, some assumptions are refined in Sect. 2.2 towards our self-stabilizing solution.

2.1.1 Emulating Shared Objects

The network includes *nodes* $\mathcal{P} = \{p_1, \ldots, p_N\}$ (processors). Each node $p_i \in P$ has access to a unique identifier i and hosts either (i) a server, (ii) a client or (iii) both a server and a client. The server has access to a storage S, which is a set of records, and the client requests the servers to use these records for updating and retrieving the latest version of the emulated shared object. The coded atomic storage CAS$(k_{threshold})$ task addresses the problem of multi-writer, multi-reader (MWMR) emulation of atomic shared memory of a single object in the above settings. The system uses erasure coding for the sake of tolerating crash failures of at most f servers (Sect. 3.1).

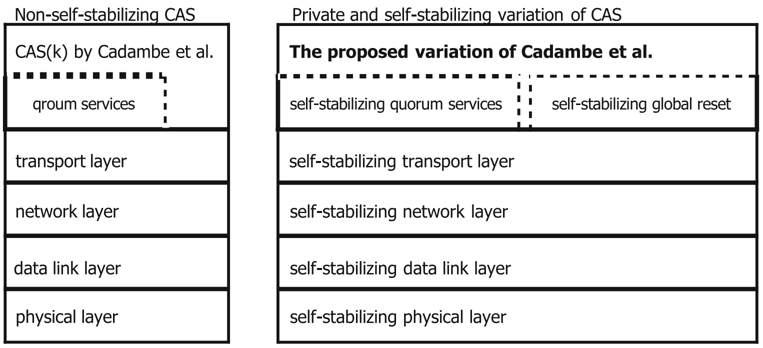
Non-self-stabilizing CAS
CAS(k) by Cadambe et al.
qroum services
transport layer
network layer
data link layer
physical layer
Private and self-stabilizing variation of CAS
The proposed variation of Cadambe et al.
self-stabilizing quorum services
self-stabilizing global reset
self-stabilizing transport layer
self-stabilizing network layer
self-stabilizing data link layer
self-stabilizing physical layer

Fig. 1 A possible deployment of the $CAS(k_{threshold})$ algorithm by Cadambe et al. [15] (on the left) and the proposed self-stabilizing variation (on the right); our contribution appears in bold

The object value is a member of a finite set $\mathcal{V}$, which $\lceil \log_2 |\mathcal{V}| \rceil$ bits can represent. We refer to $v_0 \in \mathcal{V}$ as the default (initial) state of the emulated object. A local source commands its client to run the reader or writer procedures, sequentially. A call to a reader returns the current version of the object value. A call to a writer includes the new version of the object value and returns upon completion. A writer associates each write request with a unique tag, $t \in \mathcal{T}$, where $\mathcal{T} = \mathbb{Z}^+ \times P$ and $\mathbb{Z}^+$ is the set of all positive integers. Note that $\mathcal{T}$ is a set for which the relation $<\equiv (z_1 < z_2) \vee ((z_1 = z_2) \wedge (i < j))$ can order totally any pair of tags, (z_1, p_i) and (z_2, p_j). We denote the default tag value, $t_0 < \min \mathcal{T}$, as a tag that is not in $\mathcal{T}$ and yet it is smaller than any other tag in $\mathcal{T}$.

The detailed specification of task $\text{CAS}(k_{threshold})$ [15] and [46, Chapter 13] considers each version of the object and requires the algorithm's external behavior to follow the ones of atomic (linearizable) memory. An atomic shared memory object is one where the commands to the clients and the returned values from these calls appear as if the object is being accessed sequentially rather than via concurrent calls to the client procedure. The detailed task specification requires the possibility to include in the system execution serialization points, so that the trace of the complete operation corresponds to the one of a read-write variable type. $\text{CAS}(k_{threshold})$ also requires liveness with respect to the completion of all (non-failing) operations in any (not necessarily always fair) execution in which the number of server failures is at most f, where $k_{threshold} \in \{1, \ldots, N - 2f\}$.

2.1.2 External Building Blocks

We handle node and communication failures, as well as transient faults using common external building blocks.

- *End-to-end protocols* The implementation of the system services requires the availability of an end-to-end protocol. Our self-stabilizing implementation of the services below assumes the availability of self-stabilizing end-to-end protocols, such as the ones in [22, 24]. Note that self-stabilizing end-to-end protocols assume

that the channel has a bounded capacity due to well-known impossibility results [18, Chapter 3.2].
- *Gossip services* Cadambe et al. assume the availability of a reliable gossip service. They use this non-self-stabilizing service to propagate reliably among the servers the tag of every object version. We consider a self-stabilizing gossip service (which we specify in Sect. 5.1 and propose an implementation in Sect. 6). This service lets each gossip message to overwrite the previous gossip message that is stored in the buffers (without considering whether the previous message was delivered to all receivers).

Our specifications are motivated by the fact that self-stabilizing end-to-end protocols must consider communication channels with bounded capacities [18, Chapter 3.2]. Therefore, due to the asynchronous nature of the system, a specific quorum of servers might process write operations much faster than the rate in which gossip messages arrive *reliably* to servers that are not part of that quorum. Since the communication channels are assumed to be bounded, it is not clear how can the writer avoid blocking (and still deliver all gossip messages).

- *Quorum services* Quorum systems can be used for ensuring transaction atomicity in a replica system despite the presence of network failures [55]. As explained in Sect. 1.2, a quorum system, $\mathcal{Q}$, refers to all subsets of P, such that each quorum set $Q \in \mathcal{Q}$ satisfies the quorum system specifications. We consider a quorum system that has a parameter, $k_{threshold} \in \{1 \ldots, N - 2(f + e)\}$. We require that $\lceil \frac{N+k_{threshold}+2e}{2} \rceil \leq |Q|$, where e is the maximal number of data-corrupting malicious servers and f is a bound on the ones that can crash.

Cadambe et al. [15] assume that the operations at a given client follow a "handshake" discipline, where a new invocation awaits the response of a preceding invocation. In the context of self-stabilization, this synchronization between clients and servers is subject to transient faults. Thus, we specify a service that provides this "handshake" discipline in a self-stabilizing manner (Sect. 5.1). (We offer a self-stabilizing implementation of the service in Sect. 6).

- *Reset services* In the absence of transient-faults, one can safely assume that all tags used by the algorithm are unbounded. This assumption can be made valid for any practical setting by letting each tag use a sufficient number of bits, say, 64, because counting (using sequential steps) from zero to maximum value of the tag will take longer than the time during which the system is required to remind operational. Specifically, assuming that each quorum access requires at least one nanosecond, it would take at least 584 years until the maximum value can be reached. However, in the context of self-stabilization, a single transient-fault can set the tag value into one that is close to the maximum value, which we denote by $z_{\max}$, regardless of how many finite number of bits the tag can have, where $z_{\max} \in \mathbb{Z}^+$ is a predefined positive integer. Thus, the proposed self-stabilizing solution uses bounded set of tag values, i.e., $\mathcal{T} = \{1, \ldots, z_{\max}\} \times P$. (We use the same notation of $\mathcal{T}$ for both variations whenever it is clear from the context whether the system considers self-stabilization.)

As mentioned, a single transient fault can introduce a tag value that is (close to) the maximum of $\mathcal{T}$. Therefore, there is a need to recycle obsolete tag numbers. Thus, the proposed self-stabilizing algorithm uses a self-stabilizing global reset mechanism that resembles the one considered in [6]. It helps the algorithm to overcome the case in which the system state includes a tag that is (close to) the maximum value in $\mathcal{T}$. This reset mechanism leaves the storage of every server only with the most recent version of the object and replaces its tag value with a tag that is slightly above t_0. We specify the interface between the proposed algorithm and the self-stabilizing global reset mechanism (Sect. 5.2) and note that its liveness property requires schedule fairness.

2.2 Models

Towards a self-stabilizing solution, we refine our system assumptions. Consider an asynchronous message-passing network in which the nodes can be modeled as finite state-machines that exchange messages via communication links (with bounded capacity).

2.2.1 Communication Model

The network topology is of a fully-connected graph, K_N, and any pair of nodes has access to a bidirectional communication channel that, at any time, has at most $\mathsf{capacity} \in \mathbb{N}$ packets. Every two nodes exchange (low-level messages called) *packets* to permit delivery of (high level) messages. When node $p_i \in \mathcal{P}$ sends a packet, m, to node $p_j \in \mathcal{P} \setminus \{p_i\}$, the operation $send$ inserts a copy of m to $channel_{i,j}$, while respecting the upper bound $\mathsf{capacity}$ on the number of packets in the channel. In case $channel_{i,j}$ is full, i.e., $|channel_{i,j}| = \mathsf{capacity}$, the sending-side simply overwrites any message in $channel_{i,j}$. When p_j receives m from p_i, the system removes m from $channel_{i,j}$. As long as that there is an $m \in channel_{i,j}$, we say that m's message is in transit from p_i to p_j. Recall that we assume access to a self-stabilizing end-to-end protocol [22, 24] that provides reliable (FIFO) message delivery (over unreliable non-FIFO channels that are subject to packet omissions, reordering and duplication).

2.2.2 Execution Model

Our analysis considers the *interleaving model* [18], in which the node's program is a sequence of *(atomic) steps*. Each step starts with an internal computation and ends with a single communication operation, i.e., message $send$ or $receive$.

The *state*, s_i, of $p_i \in \mathcal{P}$ includes all of p_i's variables, as well as the set of all incoming communication channels. Note that p_i's step can change s_i, as well as remove a message from $channel_{j,i}$ (upon message arrival) or add a message in $channel_{i,j}$ (when a message is sent). The term *system state* refers to a tuple of the form $c = (s_1, s_2, \ldots, s_N)$ (system configuration), where each s_i is p_i's state (including messages in transit to p_i). We define an *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ as an alternating sequence of system states c_x and steps a_x, such that each system state c_{x+1}, except for the starting one, c_0, is obtained from the preceding system state c_x by the execution of step a_x.

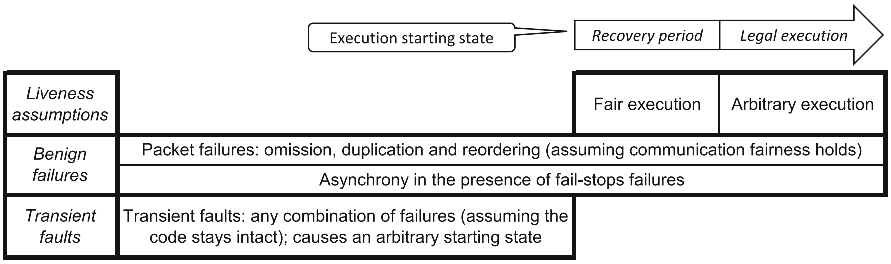
Execution starting state
Recovery period
Legal execution
Liveness assumptions
Fair execution
Arbitrary execution
Benign failures
Packet failures: omission, duplication and reordering (assuming communication fairness holds)
Asynchrony in the presence of fail-stops failures
Transient faults
Transient faults: any combination of failures (assuming the code stays intact); causes an arbitrary starting state

Fig. 2 The fault model and liveness assumptions during the system execution

Let R' and R'' be a prefix, and respectively, a suffix of R, such that R' is a finite sequence, which starts with a system state and ends with a step $a_x \in R'$, and R'' is an unbounded sequence, which starts in the system state that immediately follows step a_x in R. In this case, we can use $\circ$ as the operator to denote that $R = R' \circ R''$ concatenates R' with R''.

2.2.3 Fault Model

We model a failure as a step that the environment takes rather than the algorithm. We consider failures that can and cannot cause the system to deviate from fulfilling its task (Fig. 2). The set of *legal executions* (LE) refers to all the executions in which the requirements of the task T hold. For example, $T_{\text{CAS}(k_{threshold})}$ denotes our studied task of shared memory emulation and $LE_{\text{CAS}(k_{threshold})}$ denotes the set of executions in which the system fulfills $T_{\text{CAS}(k_{threshold})}$'s requirements. We say that a system state c is *legitimate* when every execution R that starts from c is in LE. When a failure cannot cause the system execution (that starts in a legitimate state) to leave the set LE, we refer to that failure as a benign one. We refer to any temporary violation of the assumptions according to which the system was designed to operate (as long as program code remains intact) as transient faults. We note that transient faults can cause the system execution to leave the set LE. Self-stabilizing algorithms deal with benign failures (while fulfilling the task requirements), and they can also recover after the occurrence of transient faults within a bounded period.

- *Benign failures* The algorithmic solutions that we consider are for asynchronous (message-passing) systems and thus they are oblivious to the time in which the packets arrive and depart (and require no explicit access to clock-based mechanisms, which may or may not be used by the system underlying mechanisms, say, for congestion control at the end-to-end protocol).

Communication fairness. Recall that we assume that the communication channel handles packet failures, such as omission, duplication, reordering (Sect. 2.2.1). We assume that if p_i sends a message infinitely often to p_j, node p_j receives that message infinitely often. We call the latter the *fair communication* assumption. Note that fair communication provides no bound on the channel communication delays. It merely says that a message is received within some finite time if its sender does not stop sending it (until it receives the acknowledgment message).

Node failure. We assume that the failure of node $p_i \in \mathcal{P}$ implies that its hosted client and server stops sending and receiving messages (and it also stops executing all other steps). We assume that the number of failing nodes that host servers is bounded by f and that $2f < N$; for the sake of guaranteeing correctness [46]. We bound only by N the number of nodes that host clients and fail. Moreover, failing nodes that host servers resume (without detection) within some unknown finite time and reset the server state machines by removing all stored records. However, nodes that host clients do not resume (or allow the invocation) any client process until they call a procedure that we name *localReset*(). (We specify how a global reset mechanism uses this procedure in Sect. 5.2. Moreover, Sect. 12.1 provides an elegant extension that lets nodes to recycle their client identifiers and thus the above assumption is not restrictive.)

- *Transient faults* We consider arbitrary violations of the assumptions according to which the system and the communication network design to operate. We refer to these violations and deviations as *transient faults* and assume that they can corrupt the system state arbitrarily (while keeping the program code intact). We preserve the occurrence of a transient fault as a rare event. Thus, our model assumes that the last transient fault occurred before the system execution started. Moreover, it left the system to start in an arbitrary state (while keeping the program code intact).

2.2.4 Dijkstra's Self-Stabilization Criterion

An algorithm is *self-stabilizing* with relation to the task LE, when every (unbounded) execution R of the algorithm reaches within a bounded period a suffix $R_{legal} \in LE$ that is legal. That is, Dijkstra [17] requires that $\forall R : \exists R' : R = R' \circ R_{legal} \wedge R_{legal} \in LE$, where the length of R' is polynomial in n. We say that a system *execution is fair* when every step that is applicable infinitely often is executed infinitely often, and fair communication is kept. Self-stabilizing algorithms often assume that R is a fair execution. Wait-free algorithms guarantee that non-failing operations always become (within a finite number of steps) complete even in the presence of benign failures. Note that fair executions do not consider crash failures (that were not detected by the system who then excluded these failing nodes from reentering the system). Therefore, we cannot demonstrate that an algorithm is wait-free by assuming that the system execution is always fair.

2.2.5 Self-Stabilization in the Presence of Seldom Fairness

As a variation of Dijkstra's self-stabilization criterion, we propose design criteria in which (i) any execution $R = R_{recoveryPeriod} \circ R' : R' \in LE$, which starts in an arbitrary system state and has a prefix ($R_{recoveryPeriod}$) that is fair, reaches a legitimate system state within a bounded prefix $R_{recoveryPeriod}$. (Note that the legal suffix R' is not required to be fair.) Moreover, (ii) any execution $R = R'' \circ R_{globalReset} \circ R''' \circ R_{globalReset} \circ \ldots : R'', R''', \ldots \in LE$ in which the prefix of R is legal, and not necessarily fair but includes at most $\mathcal{O}(N \cdot z_{\max})$ (Sect. 2.1.2) write operations, has a suffix, $R_{globalReset} \circ R''' \circ R_{globalReset} \circ \ldots$, such that $R_{globalReset}$ is required to be fair and bounded in length but might permit the violation of liveness requirements, i.e., a bounded number of operations might be aborted (as long as the safety requirement holds). Furthermore, R''' is legal and not necessarily fair but includes at least $z_{\max}$ write operations before the system reaches another $R_{globalReset}$. Since we can choose $z_{\max} \in \mathbb{Z}^+$ to be a very large value, say 2^{64}, and the occurrence of transient faults is very rare, we refer to the proposed criteria as one for self-stabilizing systems that their executions fairness is unrequited except for seldom periods. Next, we define how we bound the length of $R_{recoveryPeriod}$ and $R_{globalReset}$, which are the complexity measures.

2.2.6 Complexity Measures

The main complexity measure of self-stabilizing systems is the time it takes the system to recover after the occurrence of the last transient fault. In detail, in the presence of seldom fairness this complexity measure considers the maximum of two values: (i) the maximum length of $R_{recoveryPeriod}$, which is the period during which the system recovers after the occurrence of transient failures, and (ii) the maximum length of $R_{globalReset}$. We consider systems that use a bounded among of memory and thus as a secondary complexity measure, we bound the memory that each node needs to have (after considering a version that does not consider such bounds, as in Cadambe et al. [15]). However, the number of messages sent during an execution does not have an immediate relevance in the context of self-stabilization, because self-stabilizing systems never stop sending messages [18, Chapter 3.3]. Next, we present the definitions, notations and assumptions related to the main complexity measure.

- *Message round-trips* Let $c \in R$ be a state, such that immediately after c, node p_i sends a message m to p_j. Moreover, immediately after c' (that follows c), p_j receives message m (or a message that was sent from p_i to p_j after m) and sends a response message r_m back to p_i. Then, immediately after state $c'' \in R$ (that appears after c' in R), p_i receives p_j's response, r_m (or a response that was sent from p_j to p_i after r_m). If c, c' and c'' do appear in R, we say that p_i has completed with p_j a round-trip of message m.
- *Completing client rounds* A call to a client procedure results in a number of requests that the client sends to all servers and then waits for the server responses. The client may decide not to wait for responses from all servers; instead, it can decide to continue with the next phase or to reach the end of the procedure execution. We say that a client starts a new round when, after a finite period of internal computation, it sends the first request (of any phase) to the servers. Moreover, this client ends this round when it finished waiting for the server responses (and perhaps also reaches the procedure end; regardless of whether it enters branches). The client at node p_i performs a complete round when it starts a new round in $c_{start} \in R$ and ends it in $c_{end} \in R$.

We are also interested in the cases of incomplete operations, which do not have necessarily a proper start to their first round. In this case, we say that the client at node p_i completes a round when it reaches $c_{end} \in R$. Note that whenever p_i does not fail, c_{end} is well-defined, because it refers to the case in which p_i stops waiting for the server responses and moves on (to the next phase or reaching the end of the client procedure). For the case in which p_i fails, we define c_{end} to be the system state that immediately follows the step in which p_i fails.

- *Complete node and server iterations* Recall the fact that self-stabilizing algorithms can never stop communicating [18, Chapter 3.3]. The program of a self-stabilizing algorithm often includes a do-forever loop or, as in case of the proposed algorithm, a repeated gossip exchange among the servers. Next, we define the term complete iteration, which refers to such gossip exchanges.

Node complete iterations. Let $P(i) \subseteq P$ be the set of nodes with whom p_i completes a message round trip infinitely often in R. Suppose that immediately after the system state $c_{start} \in R$, node p_i takes a step that includes the execution of the first line of the do forever loop (or of the gossip procedure), and immediately after system state $c_{end} \in R$, it holds that: (i) p_i had finished the iteration that it had started in c_{begin} (regardless of whether it enters branches), and (ii) the message m_j completes its round trip, where m_j refers to any message that p_i sends during that iteration to node $p_j \in P(i)$. In this case, we say that p_i's iteration starts at c_{begin} and ends at c_{end}. *Server complete iterations.* The servers repeatedly receive messages from all other non-failing servers and then, after some internal processing, send messages to all other servers. The successful arrival of such a message to any server results again in some internal processing and then sending messages to all other servers. We say that: (i) the iteration of the server at $p_i \in \mathcal{P}$ starts when p_i first gets a message from another server, (ii) after some internal processing, p_i sends a message to every other server at p_j, (iii) this iteration continues toward letting p_j to receive that message (or a later message) from p_i; at least once, and then (iv) letting p_j's response (or a later message from p_j) to arrive at p_i and by that ending this server iteration. Given an execution R, we say that its prefix R' includes a complete iteration of the server at $p_i \in \mathcal{P}$ if R' includes p_i's iteration start and then (after that start) and p_i's iteration end appears in R'.

- *Asynchronous cycles* We measure the time between two system states in a fair execution by the number of (asynchronous) cycles between them. The definition of (asynchronous) cycles considers the term of complete iterations. The first (asynchronous) cycle (with round-trips) of fair execution $R = R'' \circ R'''$ is the shortest prefix R'' of R, such that each non-failing node and server in the network executes at least one complete iteration in R'' (which requires the exchange of messages, as specified above), where $\circ$ is the concatenation operator (Sect. 2.2.2). Moreover, each node that runs a client procedure during R'' must complete within R'' at least one client round. The second cycle in execution R is the first cycle in execution R'', and so on.

3 Background

Cadambe et al. [15] use erasure codes for emulating shared memory and use quorums to distinguish among writer, server, and reader nodes. Their algorithm allows multiple writers, using a $(N, k_{threshold})$ maximum distance separable [51] (MDS) code, to write data concurrently to the group of servers while ensuring atomicity and liveness. This section reviews the definition of $(N, k_{threshold})$ MDS code before explaining how to use them for secret sharing in a slightly adapted variation of Cadambe et al. [15].

Cadambe et al. [15] divide the data into a number of coded elements. Each server stores at most one coded element. Cadambe et al. guarantee that the reader client can fetch the necessary number of coded elements, such that the reader can retrieve the original data. Given two positive integers $m, k_{threshold} \in \mathbb{Z}^+ : k_{threshold} < m$, Cadambe et al. consider an $(m, k_{threshold})$ Maximum Distance Separable (MDS) code that maps a $k_{threshold}$-length vector (the input) to an m-length vector (the output). The aim is that after altering arbitrarily $k_{threshold}$ coordinates of the output vector, a decoding algorithm can still retrieve the input vector. This way, Cadambe et al. use an $(m, k_{threshold})$ code for storing the input vector on m servers, i.e., the server at p_i stores the output's i-th coordinate, because the decoding algorithm is resilient to $(m - k_{threshold})$ node failures. We bring the definition of $(m, k_{threshold})$ MDS code (Sect. 1) before proving the Cadambe et al.'s CAS$(k_{threshold})$ algorithm (Sect. 3.2).

3.1 Maximum Distance Separable (MDS) Codes

Let A be an arbitrary finite set and $S \subseteq \{1, 2, \ldots, m\}$. Denote by π_S the natural projection mapping from A^m onto S's corresponding coordinates, i.e., $S = \{s_1, s_2, \ldots, s_{|S|}\}$, where $s_1 < s_2 \ldots < s_{|S|}$, and define $\pi_S : A^m \rightarrow A^{|S|}$ as $\pi_S(x_1, x_2, \ldots, x_m) = (x_{s_1}, x_{s_2}, \ldots, x_{s_{|S|}})$.

Definition 1 (*Maximum Distance Separable (MDS) code*) Let A be a finite set and $m, k_{threshold} \in \mathbb{Z}^+ : k_{threshold} < m$ two positive integers. An $(m, k_{threshold})$ code over A is a map $\Phi : A^{k_{threshold}} \rightarrow A^m$. An $(m, k_{threshold})$ code Φ over A is said to be Maximum Distance Separable (MDS) if, for every $S \subseteq \{1, 2, \ldots, m\}$, such that $|S| = k_{threshold}$, there is a function $\Phi_S^{-1} : A^{k_{threshold}} \rightarrow A^{k_{threshold}}$, such that $\Phi_S^{-1}(\pi_S(\Phi(x)) = x$ for every $x \in A^{k_{threshold}}$, where π_S is the natural projection mapping.

Cadambe et al. [15] refer to each of the coordinates of the output of an $(m, k_{threshold})$ code Φ as a coded element. Further details about Φ and erasure code appear in [15]. We extend the use of $(m, k_{threshold})$ MDS code to secret sharing (Sect. 4.1).

3.2 Cadambe et al.'s CAS$(k_{threshold})$ Algorithm

Cadambe et al. [15] present a quorum-based algorithm for implementing the $CAS(k_{threshold})$ task. Algorithm 1 is our interpretation of the non-self-stabilizing $CAS(k_{threshold})$ algorithm by Cadambe et al. [15] with slight adaptations for the proposed secret sharing scheme.

3.2.1 External Building Blocks: Quorum and Gossip Communications

Cadambe et al. [15] specify $\lceil \frac{N+k_{threshold}}{2} \rceil \leq |Q|$ for any $k_{threshold} \in \{1, \ldots, N - 2f\}$ and show Lemma 1.

Algorithm 1: A non-self-stabilizing $CAS(k_{threshold})$ algorithm (that is based on Cadambe et al. with adaptations for the proposed secret sharing scheme), code for p_i's client and server.

```
1  The client: // At any time, $p_i$'s client is a writer, a reader, or none but not both $writer(s)$: ;  /* The writer
   stores secret $s$ as the new version of the shared object */
   /* Query for finalized tags and after hearing from a quorum get the maximal
      tag                                                                          */
2  let $(z, j) := \max(\{t' : (t', \bullet) \in$ qrmAccess$((\bot, \bot, \text{'qry'}))\})$;     /* Obtain coded elements
   $w_1, w_2, \ldots, w_N$, such that $p_i \in P$ has a server, by applying the $\Phi$ to the secret
   $s$. Then, send $(t, w_i, \text{'}pre\text{'})$ to every server and wait for a quorum of replies. */
3  qrmAccess$(((z+1, i), \{\Phi_{p_j}(s)\}_{p_j \in P}, \text{'pre'}))$;              /* The prewrite phase  */
   /* For each server, send $(t, \text{'}null\text{'}, \text{'fin'})$ and wait for a quorum of replies.   */
4  qrmAccess$(((z+1, i), \bot, \text{'fin'}))$;                                      /* The finalize phase  */
5  return;

6  $reader()$: ;          /* The reader retrieves the current object version, or $\bot$ upon
   failure */
7  let $t := \max(\{t' : (t', \bullet) \in$ qrmAccess$((\bot, \bot, \text{'qry'}))\})$ ;      /* Query as in line 67 */
   /* For each server, send $(t, \bot, \text{'fin'})$ and wait for a quorum of replies with
      the requested coded elements, which are associated with tag $t$.           */
8  let $Q :=$ qrmAccess$((t, \bot, \text{'fin'}))$ // Ask and wait for finalized records from a quorum /* Test whether at
      least $k_{threshold}$ replies include coded elements so that $\Phi^{-1}$ can decode the
      secret before returning it. If the test fails, return $\bot$.                 */
9  if $|\{(t, w, \text{'}fin\text{'}) \in Q : w \neq \bot\}| \geq k_{threshold}$ then return$(\Phi^{-1}(w : \{(t, w, s) \in Q : w \neq \bot\}))$;
10 else return $\bot$;

11 The server: $S \subset \mathcal{T} \times (\mathcal{W} \cup \{\bot\}) \times \mathcal{D}$ is a record set, where $\mathcal{T} = \mathbb{Z}^+ \times \mathcal{P}$ is the set of tags, $\mathcal{W}$ the set of coded
   words and $\mathcal{D} = \{\text{'pre'}, \text{'fin'}\}$ the set of phases. When $S = \emptyset$, we use the default triple $(t_0, w_{t_0,i}, \text{'fin'})$ when
   reporting on the triple with the highest locally known tag;

12 upon query arrival from $p_j$'s client to $p_i$'s server do {Reply with $(maxPhase(\text{'fin'}), \bot, \text{'qry'})$, where
   $maxPhase(\text{'fin'})$ refers to the highest tag in any record in $S$ that has a label 'fin' (whether that record includes a
   coded element or not)}
13 upon pre-write $m := (t, w, \text{'pre'})$ arrival from the $p_j$'s writer to $p_i$'s server do
14     if $\nexists(t, \bullet) \in S$ then $S \leftarrow S \cup \{(t, w, \text{'pre'})\}$; /* add the arriving record to $S$ */
15     Moreover, acknowledge the arriving record by calling reply$(j, m)$.
16 upon finalize $m = \langle t, \bot, \text{'fin'}\rangle$ arrival from $p_j$'s writer to $p_i$'s server do
17     if $\exists(t, w, \text{'}pre\text{'}) \in S$ then /* update the record $(t, w, \text{'pre'})$ to $(t, w, \text{'fin'})$ in $S$ */
18         $S \leftarrow (S \setminus \{(t, w, \text{'pre'})\}) \cup (t, w, \text{'fin'})$
19     else add $(t, \bot, \text{'fin'})$ to $S$;
20     Moreover, acknowledge to the writer by calling reply$(j, m)$ and gossip the message $(t)$ to all other servers by
       calling gossip$(t)$.
21 upon finalize $m := (t, \bot, \text{'fin'})$ arrival from $p_j$'s reader to $p_i$'s server do
22     if $\exists(t, w_i, \bullet) \in S$ then
23         $S \leftarrow (S \setminus \{(t, w, \bullet)\}) \cup \{(t, w, \text{'fin'})\}$ ; /* update the record $(t, w_i, \bullet)$ to $(t, w_i, \text{'fin'})$ in
           $S$ */
24         acknowledge the reader with $(t, w_i, \text{'fin'})$;
25     else
26         $S \leftarrow S \cup \{(t, \bot, \text{'fin'})\}$ ;                          /* add $(t, \bot, \text{'fin'})$ to $S$ */
27         acknowledge to the reader by calling reply$(j, m)$;
28     Moreover, gossip the message $(t)$ to all other servers by calling gossip$(t)$.
29 upon gossip $(t)$ arrival from $p_j$'s server to $p_i$'s server do {if $\exists(t, \bullet) \in S$ then update the record $(t, \bullet)$ to
   $(t, \bullet, \text{'fin'})$ in $S$ else add $(t, \bot, \text{'fin'})$ to $S$}
```

Lemma 1 (Lemma 1 in [15]) *Suppose that $k_{threshold} \in \{1 \ldots, N-2f\}$. (i) If $Q_1, Q_2 \in \mathcal{Q}$, then $|Q_1 \cap Q_2| \geq k_{threshold}$. (ii) If the number of failed servers is at most f, then $\mathcal{Q}$ contains at least one quorum set Q of non-failed servers.*

Algorithm 1 accesses the servers via a call to the function qrmAccess(), which returns a set of replies (records) from at least a quorum of servers. Algorithm 1 also assumes the availability of a reliable gossip service, which allows the servers to send their most recent finalized tags $t \in T$.

3.2.2 Local Variables

The state of the server includes a set of records $(t, w, label) \in S \subset \mathcal{T} \times (\mathcal{W} \cup \{\perp\}) \times \mathcal{D}$ (line 11), where the term label $d \in \{$'pre', 'fin'$\}$ refers to metadata that records the phases of the shared-object updates. The clients carry these updates sequentially; in each phase, they access the quorum system and do not end the phase before getting replies from at least a quorum. Algorithm 1 assumes that when $S = \emptyset$, the default triple $(t_0, w_{t_0,i},$ 'fin'$)$ is included in S when reporting on the triple with the highest locally known tag.

3.2.3 Protocol Phases

Both the writer and reader protocols use the query phase for discovering a recent record with the label 'fin' as its metadata (line 12). During the pre-write phase of write operations (line 13), the writer makes sure that at least a quorum of servers, say Q_{pw}, store each a coded element with the tag t' and label 'pre'. Note that immediately at the end of the prewrite phase, the stored record cannot be accessed by the readers, because when a server replies to queries it only considers records with finalized tags (line 12). However, after the prewrite phase, the writer starts the finalize phase (lines 4 and 16), which diffuses the records with the label 'fin' and the tag t' and then waits for a quorum of servers, say Q_{fw}, to reply. Immediately after this finalized phase, any query phase (of any read or write operation) will retrieve a tag that is at least as high as t' (because by Lemma 1 it holds that Q_{pw} and Q_{fw} must interest) and in that sense, tag t' is viable to all clients. Moreover, the existence of a stored record with the label 'fin' implies that the coded elements associated with tag t' are stored by at least a quorum of servers, which is Q_{pw}. This property allows the reader to retrieve at least $k_{threshold}$ unique coded elements (line 67 to 10), which are stored at the servers of Q_{pw}. Cadambe et al. [15] set the value of $k_{threshold}$ to $k_{threshold}$ (whereas we consider another value in Sect. 4). We also note that the reader further facilitates the diffusion of finalized tags to a quorum (line 8). This and the gossip messages (line 29) allow the system to complete the diffusion of finalized records in the presence of crash failures of writers.

Corollary 1 ([15], Theorem 1) *Algorithm 1 emulates a shared atomic memory for multi-writer and multi-reader.*

4 Basic Results

We present a variance that adds privacy provision to the implementation proposed by Cadambe et al. [15]. Our variation allows at most e data-corrupting malicious servers and at most f failures in an asynchronous message-passing system. In this section, we consider data-corrupting malicious servers that can send corrupted secret shares to readers, but not corrupted tags or labels, i.e., when a data-corrupting malicious server replies with a tuple (t, w, d), only w might be corrupted. Writers divide secrets and submit the resulting secret shares to the servers. Servers store their secret shares and deliver them to the readers upon request. In Sects. 2.2 to 11, we extend our proposal to withstand crash failures and server-side data-corrupting malicious behavior to also consider recovery after the occurrence of transient faults.

4.1 Using $(m, k_{threshold})$ MDS Codes for Secret Sharing

The $(N, k_{threshold})$-MDS code enables the reader to restore the data under the presence of $\frac{N-k_{threshold}}{2}$ stop-failed servers. The $(N, k_{threshold})$*-threshold scheme* for integers $k_{threshold}$ and N, such that $0 < k_{threshold} \leq N$, is defined by Shamir [54] and splits a secret s into N secret shares $\{s_i\}_{i\in\{1,\ldots,N\}}$. This scheme requires that there exists a mapping from any $S \subseteq \{s_i\}_{i\in\{1,\ldots,N\}}$ with $|S| \geq k_{threshold}$ to the secret s, but it is impossible to determine s from a set of less than $k_{threshold}$ secret shares.

Let K be a finite field, such that its size $|K|$ is a prime number. The $(N, k_{threshold})$-Reed-Solomon code, $\Phi : \mathcal{S} \to \mathcal{W}$, transforms the input data, i.e., one element of a $k_{threshold}$ dimensional vector space, $\mathcal{S}$, over K, into N dimensional vector space, $\mathcal{W}$, over the same field, K, where $k_{threshold}$ and N are as above. We call N the block length. We denote by $k_{threshold}$ the message length. The Berlekamp-Welch algorithm, Φ^{-1}, can correct $(N, k_{threshold})$-Reed-Solomon codes within $\mathcal{O}(N^3)$ time in the presence of e errors and f erasures, as long as $2e + f < N - k_{threshold} + 1$ [57], as described by Gemmell and Sudan [34]. Note that $(N, k_{threshold})$-Reed-Solomon codes are a $(N, k_{threshold})$-threshold scheme [48]. To that end, the input vector $(\sigma_1, \ldots, \sigma_{k_{threshold}}) \in \mathcal{S}$ consists of the secret σ_1 and randomly chosen values $\sigma_2, \ldots, \sigma_{k_{threshold}}$ from a uniform distribution over $\mathcal{S}$. We use Φ to map $(\sigma_1, \ldots, \sigma_{k_{threshold}})$ to the secret shares $(w_1, \ldots, w_N) \in \mathcal{W}$.

4.2 Quorums of $(k_{threshold} + 2e)$-Overlap

We require that any quorum $Q \in \mathcal{Q}$ has at least $\lceil \frac{N+k_{threshold}+2e}{2} \rceil$ servers. Lemma 2 uses the quorum definition to show that any two different quorums share at least $k_{threshold} + 2e$ servers, rather than just $k_{threshold}$ of them as in Cadambe et al. [15]. These quorums guarantee that once a writer finishes its write operation, any reader can retrieve at least $k_{threshold} + 2e$ secret shares and reconstruct the secret. The lemma also shows, similar to Cadambe et al. [15], that any two different quorums share at least $k_{threshold} + 2e$ servers. This guarantees that after a writer wrote to a quorum, the readers can read a set of coded elements that allows the secret reconstruction.

Lemma 2 (Variation of [15], Lemma 1) *Suppose that* $k_{threshold} \in \{1 \ldots, N-2(f+e)\}$. ***(1)*** *If* $Q_1, Q_2 \in \mathcal{Q}$, *then* $|Q_1 \cap Q_2| \geq k_{threshold} + 2e$. ***(2)*** *The existence of such a* $k_{threshold}$ *implies the existence of* $Q \in \mathcal{Q}$ *such that* Q *has no crashed servers.*

Proof **(1)** Let $Q_1, Q_2 \in \mathcal{Q}$, then $|Q_1 \cap Q_2| = |Q_1| + |Q_2| - |Q_1 \cup Q_2| \geq 2\left\lceil \frac{N+k_{threshold}+2e}{2} \right\rceil - N \geq k_{threshold} + 2e$. **(2)** Since there are at most f crashed servers, we can show that without such f servers, there are still enough servers alive for a quorum. It follows that $N - f \geq N - \left\lfloor \frac{N-k_{threshold}-2e}{2} \right\rfloor = \left\lceil \frac{N+k_{threshold}+2e}{2} \right\rceil$.

By Lemma 2, the atomicity and liveness analysis in [15, Theorem 5.2 to Lemma 5.9] also holds when Algorithm 1 uses $(k_{threshold} + 2e)$-overlap quorums rather than $k_{threshold}$, as Cadambe et al. [15] indented.

4.3 Privacy-Preserving Variation of Cadambe et al.

We say that a secret sharing protocol is *t-private* when a set of at most t servers cannot compute the secret, as in [8]. Note that a 0-private protocol preserves no privacy. When the presence of at most s failing servers (which do not deviate from the algorithm behavior) and at most t data-corrupting malicious servers (which deviate from the algorithm behavior only by modifying the data field of their replies to the clients), we say that the protocol is *(s, t)-robust*. This notion is similar to *t-resilience* [8].

In order to tolerate at most e (secret share corruptions made by) data-corrupting malicious servers, we propose Algorithm 1 as a variation of Cadambe et al. [15] CAS algorithm that uses $(k_{threshold} + 2e)$-overlap quorums and $(N, k_{threshold})$-Reed-Solomon codes [50], which is an $(N, k_{threshold})$-MDS [51] code that Cadambe et al. [15] use. By the atomicity and liveness analysis for the case of $(k_{threshold} + 2e)$-overlap quorums (the remark after Lemma 2), the reader retrieves $k_{threshold} + 2e$ unique secret shares with at most e manipulated shares.

4.3.1 Robustness

Robustness is added by the ability of the Berlekamp-Welch algorithm to correct errors in the Reed-Solomon codes. Note that data-corrupting malicious servers only introduce corrupted secret shares. Lemma 3 shows Algorithm 1's resilience against up to e data-corrupting malicious servers and up to f stop-failed servers.

Lemma 3 *For $k_{threshold} \in \{1 \ldots, N - 2(f + e)\}$, Algorithm 1 is (f, e)-robust.*

Proof If a writer issues a query, pre-write, and finalize operations it does not retrieve the secret from the server. Thus, writers are immune to data-corrupting malicious servers. Servers do not exchange secrets with other servers and, thus, are not directly affected by data-corrupting malicious servers.

The rest of the proof focuses on showing that when reconstructing the secret, the read operation π_r is able to be resilient against corrupted secret shares that data-corrupting malicious nodes may send. To do this, the reader queries all the servers about the maximal finalized tags and waits for a response from at least a quorum of servers. Algorithm 1 selects the maximum tag, t, for the returned set of tags. This tag t is uniquely associated with a write that reached the finalize phase before π_r's query. The read operation π_r then sends a finalize command on its own and waits for a quorum of servers to respond. Note that the reader merely collects secret shares from a quorum of servers, but never updates coded elements that the servers store, since π_r's query and finalize records only contain a $\bot$-value in place of the coded elements, which are the secret shares. By Lemma 2 and Corollary 1, it follows that any reader p_i receives at least $k_{threshold} + 2e$ secret shares from the finalize phase. Out of these $k_{threshold} + 2e$ secret shares, at most e might be corrupted. This is the case even if up to f servers are failing. Therefore, the reader can decode the secret from this collection of $k_{threshold} + 2e$ responses by applying the Berlekamp-Welch error-correction algorithm [57].

4.3.2 Privacy

Our approach ensures the privacy of the secret among servers. Lemma 4 shows that a group of less than $k_{threshold}$ servers are not able to reconstruct the secret by combining the secret shares they have stored locally.

Lemma 4 *For $k_{threshold} \in \{1 \ldots, N - 2(f + e)\}$, Algorithm 1 is $(k_{threshold} - 1)$-private.*

Proof Let t be a tag and $k_{threshold} > 1$. A set of $k_{threshold} - 1$ servers store together $k_{threshold} - 1$ secret shares associated with the tag t. Since the secret shares encode a secret using Reed-Solomon codes, it is impossible to compute the original secret with less than $k_{threshold}$ secret shares [48]. The case of $k_{threshold} = 1$ implies that the secret shares are the secret itself and, thus, privacy is compromised, i.e., it is 0-private. It follows that Algorithm 1 is $(k_{threshold} - 1)$-private.

Note that in the case of $k_{threshold} = 1$, even if privacy is not protected, it is still possible to decipher correctly corrupted secret shares. This holds because the reader blocks until it reads at least $1 + 2e$ secret shares and, thus, the additional $2e$ secret shares contain redundant information that allows the success of the Berlekamp-Welch code for error correction.

5 External Building Blocks

The proposed algorithm uses a number of external building blocks, which we specify next.

5.1 Specifications of Gossip and Quorum Services

We consider a gossip functionality that has the following interface. Servers can send gossip messages msg by calling gossip(msg). When a gossip message arrives, the receiving server raises the gossip arrival event with a set $\{gossip[k]\}_{p_k \in \mathcal{P}}$ that includes the most recently received message from every server. For the sake of simple presentation, we allow the server at node p_i to use the item $gossip[i]$ for aggregating the gossip information that it later gossips to all other servers. The gossip functionality that we consider guarantees the following: (a) every gossip message that the receiver delivers to its upper layer was indeed sent by the sender, and (b) such deliveries occur according to the communication fairness guarantees (Sect. 2.2.3). That is, our gossip service is unreliable, as opposed to the one used by Cadambe et al. [15].

We consider a system in which the clients and servers behave according to the following terms of service. At any time, any node runs only at most one client (that is either a writer or a reader). That client calls the function qrmAccess() sequentially. Moreover, the server algorithm acknowledges (by calling reply()) every request. For this client-server behavior, the quorum-based communication functionality guarantees the following. (a) At least a quorum of servers receive, deliver, and acknowledge every request. (b) The (non-failing) requesting client receives at least a quorum of these acknowledgments. (c) Immediately before the call to qrmAccess() returns, the client-side of this service clears its state from information related to the request. For the sake of simple presentation, we allow the client to call qrmAccess(msg) with two kinds of parameters; msg is either a single message to be sent to all servers, such as in the case of a query request, or a vector that includes an individual message for each server, such as in the case of a prewrite request.

We detail the above requirements in Definition 2 and use Theorem 1 (Sect. 6) in the correctness proof of the proposed algorithm.

Definition 2 (*Legal execution of the gossip and quorum services*) Let R be an execution of the algorithm that provides gossip and quorum services in which there is a client at $p_i \in \mathcal{P}$, a server at $p_j \in \mathcal{P}$ and another server at $p_k \in \mathcal{P}$.

- *Correct behavior of the gossip functionality.* Suppose that (1) every message that p_k delivers to the upper layer as a gossip from p_j was indeed sent by p_j earlier in R. Moreover, (2) such deliveries occur infinitely often in R. In this case, we say that the behavior of the gossip functionality from the server at p_j to the one at p_k is correct.
- *Terms of service for the quorum-based communication functionality.* Suppose that in R, at any time, any node runs only at most one client (that is either a writer or a reader). Moreover, that client calls the function qrmAccess() sequentially, i.e., only after the return from the call to qrmAccess() may the client call qrmAccess() again. Furthermore, suppose that the server algorithm acknowledges (by calling reply()) every request that was delivered to it. In this case, we say that R satisfies the terms of service of the quorum-based communication functionality.
- *Correct behavior of the quorum-based communication functionality.* Suppose that the client at p_i sends a request, i.e., p_i calls the function qrmAccess() in step $a_{qrmAccess} \in R$. Moreover, after $a_{qrmAccess}$, execution R includes steps (i) to (v), where (i) refers to the steps in R in which at least a quorum of servers receive $a_{qrmAccess}$'s request, (ii) refers to steps in R in which at least a quorum delivers $a_{qrmAccess}$'s request, (iii) refers to steps in R in which at least a quorum acknowledges $a_{qrmAccess}$'s request and (iv) refers to steps in R in which the client at p_i receives at least a quorum of these acknowledgments to $a_{qrmAccess}$'s request, which results in (v) a step in R in which p_i lets the function (which p_i had previously called in step $a_{qrmAccess}$) to return. Furthermore, any such return is only the result of the above sequence of steps (i) to (v). In this case, we say that the functionality of quorum-based communication is correct. In addition, immediately before

the call to qrmAccess() returns, the client-side of this service clear its state from information related to the request.

- *A legal execution of gossip and quorum services*. Let R' and R'' be a prefix, and respectively, a suffix of R, such that $R = R' \circ R''$ is an execution of gossip and quorum services that satisfies the terms of service of the quorum-based communication functionality. We say that R'' is legal when it presents: (1) a correct gossip functionality from the server at p_j to the server at p_k, and (2) a correct functionality of quorum-based communication with respect to the client at p_j.

Section 6 presents a self-stabilizing gossip and quorum services that implements that above requirements (Definition 2).

5.2 Self-Stabilizing Global Reset

The proposed algorithm uses the reset mechanism for dealing with the case in which the system includes a tag of $(z_{\max}, j) : p_j \in \mathcal{P}$ (Sect. 2.1.2). We note that the reset mechanism requires the participation of all the nodes in the network, i.e., they require execution fairness (Sect. 2.2.4). We specify the interface between the proposed algorithm and the self-stabilizing global reset mechanism.

5.2.1 The *localReset*() and *globalReset*() Functions

During an execution that is legal (with respect to the reset mechanism), the self-stabilizing global reset process starts when any node, which we refer to as the *(reset) initiator*, calls the *globalReset*(t) function; concurrent calls are allowed. The reset mechanism lets every pair of nodes exchange messages infinitely often so that it can make sure that all nodes complete the different phases of the reset process, which starts immediately after the first call to *globalReset*(). In the first phase, all client and server processes are disabled, and each node calls the function *localReset*(). In the second phase, these processes are enabled, the reset process ends, and the system resumes normal operation.

We assume that every machine, such as a server or a client, implements the function *localReset*(t). For the case of the servers, this local reset procedure removes any record from the server storage other than the ones with the tag t and then replaces the tag $t = (z, k)$ in that record with the tag $(1, k)$. Note that when $t = t_0$ (Sect. 2.1.1), no record is kept in the server storage. For the case of clients, the call to *localReset*(t) simply stops any client operation and ignores the argument t. The requirements below specify the set of legal executions. Note that the system has to reach a safe system state even when no global reset was (properly) initialized, e.g., no node has called *globalReset*(), but still, some nodes are performing reset due to transient faults.

5.2.2 Requirements

Within a bounded number of asynchronous cycles from the first step in R that includes a call to *globalReset*(t), the reset service disables all hosted processes, which are the servers and clients, and resets these processes by calling their *localReset*(t) functions, which abort all read and write operations. Moreover, every node cleans its incoming and outgoing channels, e.g., it fills these channels with reset messages so that non-reset-related messages are absent from these channels, as in [18, Chapter 3.2]. (By reset messages we mean messages that their type is only used by the reset mechanism.) Then, the reset mechanism enables every (local) machine.

We further require the following. We say that a given system state is *reset-free* when all communication channels do not include reset-related messages, and all machines (clients and servers) are enabled. Given execution R of the system, we say that R *does not include an explicit reset* when throughout R no node $p_i \in \mathcal{P}$ calls *globalReset*(). Suppose that execution R does not include an explicit reset and that all of its system states are reset-free. In this case, we say that R *does not include a spontaneous reset*. An execution R that neither contains a spontaneous reset, nor an explicit reset, is *reset-free*. When execution R does include (a spontaneous or an explicit) reset, we require R to *be done with reset* within a bounded number of Ψ asynchronous cycles. Namely, (starting from an arbitrary system state) within Ψ asynchronous cycles, the system reaches a system state after which the execution is reset-free.

5.2.3 Possible Implementations

The proposed algorithm uses a self-stabilizing global reset mechanism that resembles the one by Awerbuch et al. [6], which assume fair scheduling throughout the execution of the procedure. Another way to go is to use a self-stabilizing consensus algorithm [9], which consider a weaker design criteria for practically-self-stabilizing systems. Since similar mechanisms exist and they are not hard to extend, so that the above specifications are met, we do not consider the algorithm for implementing the specified mechanism for self-stabilizing global reset to be within the scope of this work. The interested reader can find more details about the procedure in [36, 37], which consider self-stabilizing systems that are seldom fair, and [32], which considers self-stabilizing Byzantine fault-tolerant systems that with synchrony assumptions.

6 Self-Stabilizing Gossip and Quorum Services

Algorithm 2 provides an implementation that satisfies the requirements of Definition 2. We start the algorithm description by refining the model with respect to the variables that Algorithm 2 uses, as well as its interfaces. We then detail how Algorithm 2 provides the requirements that appear in Definition 2.

6.1 Refined Model

We assume that the system has access to a self-stabilizing end-to-end (reliable FIFO) message delivery protocol [22, 24] (over unreliable non-FIFO channels that are subject to packet omissions, reordering, and duplication). The self-stabilizing algorithms in [22, 24] circulate a token between any pair of senders and receivers. Our pseudo-code interfaces that protocol via events for token departure and arrival. Moreover, the self-stabilizing algorithms in [22, 24] guarantee that the receiver raises eventually a token arrival event with a token that the sender had transmitted upon its previous token departure event; exactly one token exists at any time that succeeds the recovery period.

6.1.1 Variables

The gossip servers require two kinds of buffers (line 30): one for the messages that go out ($gossipTx$) and another for the ones that come in ($gossipRx[]$). Node p_i stores in $gossipTx$ the needed-to-transmit messages and in the buffers $gossipRx[j] : j \neq i$ the ones that are arriving to p_i from p_j's gossip. We use $gossipRx[i]$ for aggregating these received values.

Four kinds of buffers facilitate the quorum-based communications, because this communication pattern has a round-trip nature that a client initiates and includes a number of nodes (line 31). The initiating client writes the message to $pingTx$. The end-to-end communication protocol [22, 24] transfers that message to the server-side and stores it in $pongTx[]$. The server processes the arriving messages and stores its reply in $pongRx[]$ so that the end-to-end protocol could transfer this reply back to the client-side, which stores it in $pingRx[]$. When any of these buffers do not store a message, the $\perp$ symbol is used. Note that the client requests take the form of $(tag, word, phase) \in F = (\mathcal{T} \cup \{\perp\}) \times (\mathcal{W} \cup \{\perp\}) \times (\mathcal{D} \cup \{\text{‘qry’}\})$, where $phase =$ ‘qry’ whenever the client sends a query (rather than another phase that does appear in $\mathcal{D}$). The reply has the form of $(ping, pong) \in F \cup \{\perp\} \times F \cup \{\perp\}$. We use the variable $aggregated$ for letting the client aggregate the server responses for the latest request (line 32).

6.1.2 The Interface that Algorithm 2 Assumes to be Available

Every pair of nodes maintains a pair of self-stabilizing communication channels [22, 24], i.e., one channel in each direction, which circulates a token between the sender and the receiver. The code interfaces that protocol via events for token departure (lines 40, 43 and 44) and arrival (lines 41, 47, and 51). Moreover, this protocol guarantees that the receiver raises eventually a token arrival event with a token that the sender had transmitted upon its previous token departure event. Exactly one token exists at any time. In the code, token arrival and departure events raise the respective events according to the message type, which is either gossip or quorum. The event handlers to these events perform local operations, release the token by calling $send(receiverID, Payload)$, as well as raising other local events on the calling node. Algorithm 2 also assumes access to a primitive, which we call $suspend()$. When an event calls $suspend(var, const)$, the system suspends the calling event until $var = const$ (while allowing other non-suspended events on that node to run).

6.2 The Details of Algorithm 2

The gossip functionality simply sends the message in $gossipTx$ whenever the token arrives at the client (line 40), stores it in $gossipRx[j]$ whenever the token arrives at the server (line 41), and raises the gossip arrival event at the server-side with all the most recently arrived gossip messages from each server (line 42). Note that when a gossip message arrives, the receiving server raises an event with $\{gossip[k]\}_{p_k \in \mathcal{P}}$, which includes the most recently received messages. We allow a simpler presentation of Algorithm 3 by letting the server at node p_i use the item $gossip[i]$ for aggregating the gossip information that it later gossips to all other servers.

The ping-pong protocol is inspired by the Communicate protocol proposed by Attiya el at. [5, Section 3]. The client sends its message to the server upon token departure (line 43), update the server buffer upon token arrival (line 46), and then the server sends its reply upon its token departure (line 46) before the token arrival event allows the client to accumulate the server replies (line 47). The client then tests whether it has accumulated replies from a quorum of servers (line 53). If this is the case, it lets the calling client procedure receive the accumulated replies (line 55).

Note that, for the sake of simple presentation of Algorithm 3, we allow the client to call qrmAccess(msg) either when msg is a single message to be sent to all servers or when msg is a vector that includes an individual message for each server (line 43 and line 59).

6.3 Correctness of Algorithm 2

Theorem 1 shows that the system always reaches a legal execution (Definition 2) and uses Remark 1.

Remark 1 Recall that we assume the use of a self-stabilizing communication channel between any pair of nodes, such as in [22, 24], guarantees reliable end-to-end message delivery. In [22, 24], the receiver raises eventually a token arrival event with a message that the sender had transmitted upon its previous token departure event, such that, at any time, there is exactly one token carrying one message. And, the token traversal direction alternates, i.e., go from one peer to another, then back, then go again, and so on.

Theorem 1 shows the satisfaction of the requirements of Definition 2.

Theorem 1 (Self-stabilizing gossip and quorum-based communications) *Let R be an Algorithm* 2's (unbounded) execution that satisfies the terms of service of the quorum-based communication functionality. Suppose that R is fair and its starting system state is arbitrary. Within $\mathcal{O}(1)$ asynchronous cycles, R reaches a suffix R' in which ***(1)*** the gossip, and ***(2)*** the quorum-based communication functionalities are correct. ***(3)*** During R', the gossip and quorum-based communication correctly complete their operations within $\mathcal{O}(1)$ asynchronous cycles.

Proof **Part (1).** Let $a_{\text{gossip},\ell} \in R$ be a step in which the server at $p_j \in \mathcal{P}$ calls gossip(m_ℓ) for the ℓ-th time in R (line 36). Note that $a_{\text{gossip},\ell}$ copies m to $gossipTx_i$ (line 36). Let $a_{depart,\ell'} \in R$ be the first step in R that appears after $a_{\text{gossip},\ell'} : \ell' \in \{1, \ldots\}$ and before $a_{\text{gossip},\ell'+1}$, if there is any such step, in which p_j executes the event of gossip token departure (line 40). Note that $a_{\text{depart},\ell'}$ transmits to the server at $p_k \in \mathcal{P}$ the token $m_{\ell'}$, where $m_{\ell'} = gossipTx_i[i]$ in any system state that is between

Algorithm 2: Self-stabilizing gossip and quorum-based communication, code for p_i.

```
30 Variables: $gossipTx$ and $gossipRx[]$ are buffers, where $p_i$'s server stores in $gossipTx$ the needed to transmit a
   message. In $gossipRx[j] : j \neq i$, $p_i$'s server stores the most recently received $p_j$'s gossip and allows the use of
   $gossipRx[i]$ for aggregating these received values.
31 $pingTx$, $pingRx[]$, $pongTx[]$ and $pongRx[]$ are buffers, where $p_i$'s client stores in $pingTx$ its request, or $\perp$. In
   $pingRx[j]$, $p_i$'s server stores the most recently received $p_j$'s request, or $\perp$. In $pongTx[j]$, $p_i$'s server stores the
   reply to $p_j$, or $\perp$. In $pongRx[j]$, $p_i$'s client stores the most recently received $p_j$'s acknowledgment, or $\perp$. The
   client request and $pingTx$ has the form $(tag, word, phase) \in F = (\mathcal{T} \cup \{\perp\}) \times (\mathcal{W} \cup \{\perp\}) \times (\mathcal{D} \cup \{\text{'qry'}\})$. The
   reply form is $(ping, pong) \in F \cup \{\perp\} \times F \cup \{\perp\}$.
32 $aggregated$ is a variable in which $p_i$'s client stores the collected server responses for the latest request;
33 Interface in use: Every pair of nodes maintain a self-stabilizing communication channel [22, 24], which circulates
   a token between the sender and the receiver. Token arrival and departure events raise the respective events
   according to channel type, which is either gossip or quorum;
34 $suspend(var, const)$ suspends the calling event until $var = const$ (while allowing other events to run);
35 Interface provided:
36 function gossip($msg$) do $gossipTx \leftarrow msg$;
37 function qrmAccess($msg$) do $phaseInit(msg)$; return($wait()$);
38 function reply($j, m$) do {if $pingRx[j].phase =$ 'qry' then $pongTx[j] \leftarrow (m.tag, \perp, \text{'qry'})$ else $pongTx[j] \leftarrow$
   $(pingRx[j].tag, m.word, pingRx[j].phase)$}
39 Event handlers and local functions:
40 upon gossip token departure from $p_i$'s server to $p_j$'s server do send($j, gossipTx$);
41 upon gossip token $m$ arrival from $p_j$'s server to $p_i$'s server do
42     $gossipRx[j] \leftarrow m$; raise the gossip arrival event with the message $\{gossipRx[k]\}_{p_k \in \mathcal{P}}$;
43 upon pingpong token departure from $p_i$'s client to $p_j$'s server do send($j, load(j, pingTx)$);
44 upon pingpong token departure from $p_i$'s server to $p_j$'s client do
45     if $pingRx[j] = \perp$ then send($j, (\perp, \perp)$); /* ignore the server during channel resets */
46     if $pingRx[j] \neq \perp$ then send($j, (pingRx[j], pongTx[j])$);
47 upon $pingpong$ token arrival from $p_j$'s client to $p_i$'s server do
48     $pingRx[j] \leftarrow pingpong$;                         /* $pingpong$ is either $\perp$ or $msg$ */
49     if $pingRx[j] \neq \perp$ then /* don't interrupt the server during channel resets */
50         raise the event of $pingRx[j].phase$ arrival from $p_j$'s server with $pingRx[j].msg$ (unless it is $\perp$)
51 upon $pingpong = (ping, pong)$ token arrival from $p_j$'s server to $p_i$'s client do
52     if $load(j, pingTx) = ping \land (pong = \perp \lor pong.tag = \perp \lor ((ping.phase \neq \text{'qry'}) \implies (ping.tag =$
       $pong.tag)))$ then $pongRx[j] \leftarrow pong$;
53     if $\{p_j : pongRx[j] \neq \perp\} \in \mathcal{Q}$ then /* test for a quorum of acknowledgments */
54         $aggregated \leftarrow \{pongRx[j].word : pongRx[j] \neq \perp\}$;
55         $clear()$; /* returns from $ping.phase$ with $aggregated$ as the acknowledgment set
           */
56 function $clear()$ do begin foreach $p_k \in \mathcal{P}$ do $pongRx[k] \leftarrow \perp$ end; $pingTx \leftarrow \perp$;
57 function $phaseInit(m)$ do begin foreach $p_k \in \mathcal{P}$ do $pongRx[k] \leftarrow \perp$ end; $pingTx \leftarrow m$;
58 function $wait()$ do $suspend(pingTx, \perp)$; let $x = aggregated$; $aggregated \leftarrow \emptyset$; return($x$);
59 function $load(j, m)$ do if $m := ((\bullet, w_1, \bullet), \ldots, (\bullet, w_N, \bullet))$ then return $(\bullet, w_j, \bullet)$ else return $m$;
```

$a_{\text{gossip},\ell'}$ and $a_{depart,\ell'}$. That token arrives eventually to the server at p_k, which raises the respective event (line 41) and then the event of gossip arrival with the message $m_{\ell'}$ (line 42). Thus, the gossip functionality is correct (Definition 2), because (1) $m_{\ell'}$ was indeed sent by p_j, and (2) p_k delivers such message infinitely often.

Correct behavior of the gossip functionality. Suppose that (1) every message that p_k delivers to the upper layer as a gossip from p_j (line 41) was indeed sent by p_j earlier in R (line 36). Moreover, (2) such deliveries occur infinitely often in R. Furthermore, (3) at any time, the communication channel from p_j to p_k does not include a message that p_j has never sent. In this case, we say that the behavior of the gossip functionality from the server at p_j to the one at p_k is correct.

Part (2).

Claims 1 and 2 imply the proof of this part.

Claim 1 *Suppose that the client at p_i sends a request, i.e., p_i calls the function qrmAccess(m) (line 37) in step $a_{qrmAccess} \in R$, where $m \neq \bot$. After $a_{qrmAccess}$, execution R includes steps (i) to (v) (Definition 2).*

Proof In $a_{qrmAccess}$, node p_i assigns $m_i \neq \bot$ to $pingTx_i$ (line 37) and then suspends the running client (while allowing other events to run concurrently) until $pingTx_i = \bot$, where $m_i \in F$. Recall that only the client calls the function qrmAccess() (Algorithm 3) and it does so sequentially (terms of service for the quorum-based communication functionality, Definition 2). Thus, after $a_{qrmAccess}$, the only way in which p_i assigns $\bot$ to $pingTx_i$ is by executing line 55. Therefore, the invariant of $pingTx_i = m_i \neq \bot$ holds until the client at p_i takes the step (v), which returns from the function that it had called in step a_{depart} with $aggregated$ as the acknowledgment set. We show that after $a_{qrmAccess}$, the system takes steps (i) to (v).
Steps (i) and (ii). Let a_{depart} be the first step in R that appears after $a_{qrmAccess}$ and in which p_i executes the event of ping-pong token departure from the client at p_i to the server at p_j (line 43). Note that p_i transmits the message $m_i = pingTx_i$ in a_{depart}. That token arrives eventually to the server at p_j (Remark 1), which raises the respective event at some step $a_j \in R$ (line 47) and then the event of $pingRx[j].phase$ arrival (also in step a_j), which delivers the arriving token, $pingRx[j].msg$ (line 50), unless the latter is $\bot$. Let $\hat{Q}_{(i),(ii)} \subseteq P$ include the set of nodes, such as p_j, that their servers raise the latter two events (in step a_j). Note that, as long as the invariant of $pingTx_i = m_i \neq \bot$ holds, $\hat{Q}_{(i),(ii)}$ includes more and more nodes. Thus, $\hat{Q}_{(i),(ii)} \in \mathcal{Q}$ eventually (Property (2) of Lemma 2).
Steps (iii). Recall that this lemma assumes that in R the server at $p_j \in \mathcal{P}$ acknowledges (by calling reply(msg), line 38) requests that p_j delivers to it (terms of service for the quorum-based communication functionality, Definition 2). Let $a_{j'} \in R$ refer to these steps and $\hat{Q}_{(iii)} \subseteq P$ include the set of nodes, such as p_j, that take these steps, $a_{j'}$. Note that, as long as the invariant of $pingTx_i = m_i \neq \bot$ holds, $\hat{Q}_{(iii)}$ includes more and more nodes. Thus, $\hat{Q}_{(i),(ii),(iii)} \in \mathcal{Q}$ eventually (Property (2) of Lemma 2).
Steps (iv) and (v). The server at p_j eventually sends the token $(pingRx_j[i], pongTx_j[i])$ (line 46), cf. Remark 1. Note that the sent token includes (in the ping field) $a_{qrmAccess}$'s request and (in the pong field) the same phase and tag of the arriving request (line 38), if the sent request includes any such values. Moreover, that token arrives eventually from the server at p_j to the client at p_i (Remark 1) at some step $a_{j''} \in R$, which raises the respective event (line 51). Recall that node p_i cannot change the value of $pingTx_i \neq \bot$ before the if-statement condition of line 52 holds. By the fact that the arriving token includes the same phase and tag, if there is any, of the sent request $pingTx_i$, node p_i collects the arriving acknowledgments until the if-statement condition of line 52 holds eventually. Let $\hat{Q}_{(iv)} \subseteq P$ be the set of nodes, such as p_j, for which the p_i takes the step $a_{j''}$. Note that, as long as the invariant of $pingTx_i = m_i \neq \bot$ holds, $\hat{Q}_{(iv)}$ includes more and more nodes. Thus, $\hat{Q}_{(iv)} \in \mathcal{Q}$ eventually (Property (2) of Lemma 2). This implies that p_i also takes the step $a_i \in R$ in which p_i lets the function (which p_i had previously called in step $a_{qrmAccess}$) to return

(line 55) by calling $clear()$ (line 55) and by that allowing the resume of the client and the return from the function that has sent the request. Only then does the invariant $pingTx_i \neq \bot$ stops from holding. Recall that $aggregated$ holds the set of server replies that were sent between $a_{qrmAccess}$ and a_i, as well as matched p_i's request, and used the same phase and tag (if there were any such values in p_i's request).

Claim 2 *Suppose that the state of the client at p_i includes a non-$\bot$ value in $pingTx_i$. Eventually, $pingTx_i = \bot$ (and the client at p_i resumes, if it had been suspended).*

Proof Suppose that the client at p_i calls eventually the function qrmAccess() (line 37). Then, by Claim 1 the proof of this claim is done.

Suppose that, throughout R, the client at p_i does not call the qrmAccess() function and yet $pingTx_i \neq \bot$ (in R's starting system state). We show that $pingTx_i = \bot$ eventually. Recall that $pingTx \neq \bot$ has the form of $(tag, \bullet, phase) \in F$, where $F = (\mathcal{T} \cup \{\bot\}) \times \{\bullet\} \times (\mathcal{D} \cup \{\text{‘qry’}\})$ and the replies have the form of $(ping, pong) \in F \cup \{\bot\} \times F \cup \{\bot\}$ (line 31). Note that eventually when p_j sends a token back to p_i (line 44). That token is either $(\bot, \bot)$ or (pin_j, pon_j), where pin_j is the request $pingRx_j[i]$ that p_j had received from p_i and pon_j is $(pingRx[j].tag, \bullet, pingRx[j].phase)$ (line 38). By the same arguments that appear in the proof of Claim 1, this proof is done. Namely, as long as $pingTx \neq \bot$ we have $pin_j \neq \bot$ and we can apply ‘steps (iv) and (v)’ in the proof of Claim 1. □

Part (3). By the proof of Part (1) of this lemma, we see that the correctness invariant of the gossip service holds within $\mathcal{O}(1)$ asynchronous cycles because it considers the propagation of a single message from p_j to every $p_k \in \mathcal{P}$, i.e., it requires a single complete server iteration (with round-trips). By the proof of Part (2) of this lemma, we see that the correctness invariant of the quorum-based communication service holds within $\mathcal{O}(1)$ asynchronous cycles because steps (i) and (v) consider the propagation of a single message round-trip from a client to a quorum of servers, i.e., it requires a single complete client round.

7 An Unbounded Self-Stabilizing CAS Algorithm

This paper presents a self-stabilizing algorithm that uses a bounded amount of memory. For the sake of presentation simplicity, we start by presenting a self-stabilizing algorithm that has no such bounds as a ‘first attempt’. We then prove the correctness of the unbounded algorithm (Sect. 8) before bounding the amount of storage needed (Sect. 9), as well as the number of possible tag values (Sect. 10).

One of the key differences between self-stabilizing algorithms to non-self-stabilizing algorithms is that, due to a transient fault, the self-stabilizing system can start in a state c that the non-self-stabilizing system can never reach. For example, in c a single server may include a record with a finalized tag t for which there is no quorum of servers that store records that include coded elements relevant to t. Due to the asynchronous nature of the system, we cannot bound the number of write operations that the system will take until at least one write operation installs its records on all servers. Similar examples can be found when considering pre-write records. We

Algorithm 3: Private and Unbounded Self-Stabilization CAS, code for p_i's client and server.

```
60 The client: //At any time, $p_i$'s client is a writer, a reader, or none but not both $writer(s)$: /* Store the
       secret $s$ as a new version of the shared object */
   /* Query for finalized tags and after hearing from a quorum get the maximal
       tag */
61 let $(z, j) := \max(\{t' : (t', \bullet) \in$ qrmAccess$((\bot, \bot, \text{'qry'}))\})$
62 qrmAccess$(((z+1, i), \{\Phi_{p_j}(s)\}_{p_j \in \mathcal{P}}, \text{'pre'}))$; /* Prewrite and wait for a quorum of replies
   */
63 qrmAccess$(((z+1, i), \bot, \text{'fin'}))$;       /* Finalize and wait for a quorum of replies */
64 qrmAccess$(((z+1, i), \bot, \text{'FIN'}))$;       /* FINALIZE and wait for a quorum of replies */
65 return;

66 $reader()$: ;      /* The reader retrieves the current object version, or $\bot$ upon
   failure */
67 let $t := \max(\{t' : (t', \bullet) \in$ qrmAccess$((\bot, \bot, \text{'qry'}))\})$ ;      /* Query as in line 67 */
68 let $Q :=$ qrmAccess$((t, \bot, \text{'fin'}))$ // Ask and wait for finalized records from a quorum if
   $|\{(t, w, \textit{'fin'}) \in Q : w \neq \bot\}| \not\geq k_{threshold}$ then return $\bot$; /* Test the number of replies */
69 else return$(\Phi^{-1}(w : \{(t, w, s) \in Q : w \neq \bot\}))$; /* Use the retrieved shares for decoding */

70 The server:
71 $S \subset \mathcal{T} \times (\mathcal{W} \cup \{\bot\}) \times \mathcal{D}$ is a record set, where $\mathcal{T} = \mathcal{Z} \times P$ is the set of tags, $\mathcal{W}$ the set of coded words and
   $\mathcal{D} = \{\text{'pre'}, \text{'fin'}, \text{'FIN'}\}$ the set of phases. When $S = \emptyset$, we use the default triple $(t_0, w_{t_0,i}, \text{'fin'})$ when reporting
   on the triple with the highest locally known tag;

72 Event handlers at the server:
73 upon query arrival from $p_j$'s client to $p_i$'s server do
74     if $p_j$'s client is a reader then reply$(j, (maxPhase(\mathcal{D} \setminus \{\text{'pre'}\}), \bot, \text{'qry'}))$;
75     else reply$(j, (maxPhase(\mathcal{D}), \bot, \text{'qry'}))$;

76 upon pre-write $(t, w, \text{'pre'})$ arrival from the $p_j$'s writer to $p_i$'s server do
77     $updatePhase(t, w, \text{'pre'})$;
78     reply$(j, (t, \bot, \text{'pre'}))$;

79 upon finalize or FINALIZE $m := (t, \bot, d) : d \in (\mathcal{D} \setminus \{\text{'pre'}\})$ arrival from $p_j$'s client to $p_i$'s server do
80     $updatePhase(t, \bot, d)$;
81     if $\exists s := (t, w, d) \in S$ and $p_j$'s client is a reader then $reply(j, (t, w, d))$ else $reply(j, (t, \bot, d))$;

82 upon gossip $\{(pre[k], fin[k], FIN[k]) = gossip[k]\}_{p_k \in \mathcal{P}}$ arrival from $p_j$'s server to $p_i$'s server do
83     $pre[i] \leftarrow \max(\{pre[k], fin[k], FIN[k]\}_{p_k \in \mathcal{P}} \cup \{maxPhase(\mathcal{D})\})$;
84     $updatePhase(pre[i], \bot, \text{'pre'})$;
85     $fin[i] \leftarrow \max(\{fin[k], FIN[k]\}_{p_k \in \mathcal{P}} \cup \{maxPhase(\mathcal{D} \setminus \{\text{'pre'}\})\})$;
86     $updatePhase(fin[i], \bot, \text{'fin'})$;
87     $FIN[i] \leftarrow \max(\{FIN[k]\}_{p_k \in \mathcal{P}} \cup \{maxPhase(\{\text{'FIN'}\})\}) \cup \{t \in \mathcal{T} : \{p_k \in \mathcal{P} : fin[k] = t\} \in \mathcal{Q}\}$;
88     $updatePhase(FIN[i], \bot, \text{'FIN'})$;
89     gossip$(tagTuple())$;

90 Local functions at the server:
91 function $maxPhase(phs)$ do return $\max(\{t : (t, \bullet, p) \in (S \cup \{(t_0, w_{0,i}, \textbf{'fin'})\}) \wedge p \in phs\})$
92 function $tagTuple()$ do return $(maxPhase(\mathcal{D}), maxPhase(\mathcal{D} \setminus \{\text{'pre'}\}), maxPhase(\{\text{'FIN'}\}))$;
93 function $updatePhase(t, w, u)$ do {if $\exists s := (t, w', c) \in S \wedge w' \neq \bot \wedge w = \bot$ then $S \leftarrow (S \setminus \{s\}) \cup \{(t, w', p)\}$),
   where $p := upgradePhase(c, u)$ else $S \leftarrow ((S \setminus \{(t, \bullet)\}) \cup \{(t, w, u)\})\}$;
94 function $upgradePhase(old, new)$ do switch $(old, new)$ :
95   case('pre', 'fin'): return 'fin'; case('fin', 'FIN'): return 'FIN'; default return $old$;
```

propose to overcome this challenge by letting the gossip server exchange a message that includes the maximal tag values for each phase.

There is no self-stabilizing algorithm for end-to-end communication when there is no bound on the capacity of the communication channels [18, Chapter 3]. Cadambe et al. [15] assume that all communication channels are reliable and cannot lose messages. We are not aware of a straightforward manner in which we can assume that these communication channels are both of bounded capacity and self-stabilizing, because the asynchronous nature of the system implies that there is no bound on the number of write operations that the system may finish before a given server receives a single gossip message. Therefore, we let the gossip service repeatedly exchange among the servers their maximal tag values. This way, the servers get to know eventually about the highest tag values.

With these modifications in mind, we note that the client part of Algorithm 3 follows similar lines as the ones of Algorithm 1 with the following notable differences. The prewrite phase (line 62) associates the operation with the tag $(z+1, i)$, where $t = (z, \bullet)$ is the maximal prewrite tag returned from the query phase (line 61). Moreover, Algorithm 3 uses an additional finalized phase (line 64), which we refer to as FINALIZED. This phase helps the algorithm to assure that every complete write operation with tag t has at least a quorum of servers with a finalized tag t.

The server part of Algorithm 3 also implements the above modifications. The notable changes, with respect to Algorithm 1, include the following. Servers reply to queries from readers with the highest local finalized tag (line 75), whereas for the case of writers, all local tags are considered (line 74). Also, upon gossip arrival (line 82), the server processes all the gossip messages that have recently arrived for all servers. It first calculates the local maximal prewrite tag (lines 83 and 84), then the local maximal finalized tag (lines 85 and 86) before considering the FINALIZED one (lines 87 and 88) and then sending an updated gossip message (line 89). Note that each server stores the highest tag that it has heard from each phase. Moreover, when a server discovers that it knows about a quorum of servers in which each server stores a finalized record with tag t, it updates that record to have the FINALIZED phase (line 87). This way, an implicit FINALIZED record becomes explicitly FINALIZED.

8 Correctness Proof of our Self-Stabilizing CAS Algorithm

After the preliminaries (Sect. 8.1), we study the basic properties of Algorithm 3 (Sect. 8.2) before showing its ability to recover after the occurrence of transient-faults (Sect. 8.3). We then demonstrate the atomicity (Sect. 8.4) and liveness (Sect. 8.5) of Algorithm 3.

8.1 Notation and Definitions

We refer to the values of variable X at node p_i as X_i, i.e., the variable name with a subscript that indicates the node identifier. We denote the storage variable, S, of p_i as S_{p_i} due to its centrality to the system state. Let R be an execution, $c \in R$ a system state, and $p_i \in \mathcal{P}$ a node that executes the function $f()$ in a step $a \in R$ that appears in R immediately after c. We denote by $f_a()$ the value that returns from $f()$'s execution during step a.

Each client procedure includes a finite sequence of requests that the client sends to the servers, where the responses received from one request to the servers are used for forming the next request to the servers. We associate each invocation of the client procedures with an operation π that includes all of its steps in R (either taken by the client or by the servers) in which a node sends or receives messages due to π's invocation of the client procedure. Definition 3 classifies operations by the way they start and end.

Definition 3 (*Classifying operations by their start and end*) Let R be the algorithm execution with π as a client operation. Denote by $c_{start}(\pi) \in R$ the system state that is followed immediately by step $a_{start}(\pi)$ that starts π. Moreover, $c_{end}(\pi)$ denotes the system state that follows immediately after a step $a_{end}(\pi)$ that ends π. We characterize π's behavior in R in the following manner.

- *Incomplete operations.* Suppose that π's first step, $a_{start}(\pi)$, does not include the execution of the first line of the π's (write or read) procedure (lines 61, and respectively, 67). In this case, we say that π is *incomplete* in R. We say that a client request or a server reply is incomplete if it is part of an incomplete operation (due to stale information that appears in an arbitrary starting system state). Note that the expression "operation π is complete in suffix R'' of R" refers to the case in which $a_{start}(\pi) \in R'$ and $a_{end}(\pi) \in R''$, where $= R' \circ R''$.
- *Failed operations.* Suppose that π's last step, $a_{end}(\pi)$, does not include the execution of the last line of the π's (read or write) procedure. In this case, we say that π *fails* in R.
- *Complete operations.* Suppose that π is eventually neither incomplete, nor failed in R. In this case, we say that π is *complete* in R. Before the end of a given complete operation, we refer to it as an *ongoing* operation.

8.2 Basic Properties of Algorithm 3

Lemma 5 (Algorithm 3's progression) *Algorithm 3's operations, whether they are failed, incomplete or complete, end within $\mathcal{O}(1)$ asynchronous cycles in any fair execution of Algorithm 3, which may start in any system state.*

Proof The server part of Algorithm 3 includes only non-blocking responses to Algorithm 3's requests. Thus, Algorithm 3's termination depends on the termination of each client phase. We, therefore, prove that every client phase ends eventually.
The query, pre-write and finalize phases (of readers and writers) terminate. We start by showing that at least a quorum of query responses arrive at every non-failing client (lines 61 and 67). The proof uses parts (2) and (3) of Theorem 1 for showing the correct functionality of quorum-based communication in R (Sect. 5.1), as long as the system satisfies the terms of service of the quorum-based communication functionality. To that end, we need to show that: (i) only the client calls the function qrmAccess() and it does so sequentially, as well as (ii) the server algorithm acknowledges (by calling reply(), Sect. 5.1) requests that were delivered to it. From Sect. 2.1.1 and Algorithm 3, we observe that: (i.a) any node run at most one client that is either a writer or a reader, (i.b) only the clients call the function qrmAccess(), (i.c) there is only one client (either reader or writer) per node, and (i.d) that client does not call qrmAccess() before the previous call returns. We also note that (ii) the server pseudo-code (Algorithm 3) includes a response for every client request. In detail, any non-failing server, say, the one at node $p_j \in \mathcal{P}$, replies to queries that p_i delivers to it with the message $((\bot, \bot, \text{'qry'}), (t, \bot, \text{'qry'}))$ (line 73), where $(t, \bullet, \text{'fin'}) \in S_{p_i}$

and t is S_{p_i}'s highest finalized local tag. Note that whenever $S_{p_i} = \emptyset$, the server at p_i considers the tuple $(t_0, \perp, \text{'qry'})$ (line 71), and thus the server at p_j always replies to queries. From (i.a), (i.b), (i.c), (i.d), and (ii) we get that the functionality of quorum-based communication is correct eventually even when starting from any system state, because the conditions of parts (2) and (3) in Theorem 1 hold (terms of service for the quorum-based communication functionality, Sect. 5.1). Therefore, the client at p_i receives eventually at least a quorum of server responses (Sect. 5.1). Using the same arguments as above, this proof shows that at least a quorum of pre-write and finalize responses arrive at every non-failing client (lines 62, 63 64, and 68).

Finally, we show that the above happens within $\mathcal{O}(1)$ asynchronous cycles. This is because each client operation considers a constant number of phases. Recall that each phase is associated with a client round, which completes within $\mathcal{O}(1)$ asynchronous cycles (Theorem 1).

Definition 4 (*Classifying local maximal tags by their phase*) Let R be an Algorithm 3's execution, and $a_{i,k} \in R$ a step (that the server at $p_i \in \mathcal{P}$ takes) in which p_i executes the function $maxPhase(phs)$ (line 91) for the k-th time in R. We characterize $maxPhase_{a_{i,k}}(phs)$'s behavior in R according to its argument phs (Lemma 6) and consider the set $tags(C, D) = \{t : (t, \bullet, d) \in (S_{p_j} \cup \{(t_0, w_{0,i}, \text{'fin'})\}) \land d \in D \land p_j \in C\}$, where $D = phs$ and $C = \{p_i\}$ in the system state that immediately precedes $a_{i,k}$.

- A *write maximal tag* is the returned value from $maxPhase_{a_{i,k}}(\mathcal{D})$ (lines 75 and 83), where $\mathcal{D} = \{\text{'pre'}, \text{'fin'}, \text{'FIN'}\}$, i.e., the tag in the maximal tuple in $tags(\{p_i\}, \{\text{'pre'}, \text{'fin'}, \text{'FIN'}\})$.
- A *read maximal tag* is the returned value from $maxPhase_{a_{i,k}}(\mathcal{D}\setminus\{\text{'pre'}\})$ (lines 74 and 85), i.e., the tag in the maximal tuple in returned from $tags(\{p_i\}, \{\text{'fin'}, \text{'FIN'}\})$.
- An *anchor maximal tag* is the returned value from $maxPhase_{a_{i,k}}(\{\text{'FIN'}\})$ (line 87), i.e., the tag in the maximal tuple in $tags(\{p_i\}, \{\text{'FIN'}\})$).

Lemma 6 (Servers do not remove their maximal records) *Servers (Algorithm 3) keep in their storage the currently maximal (1.1) write, (1.2) read, and (1.3) anchor records (or any record with a tag that is higher than the ones in these records).*

Proof **Part (1.1).** We show that the server (Algorithm 3) at p_i does not remove from S_{p_i} the maximal anchor record (Definition 3), i.e., the tuple with the maximal tag in $tags(S_{p_i}, \mathcal{D})$, cf. Definition 4. Note that the server at p_i updates and inserts records $(t, \bullet)$ to S_{p_i} only via the $updatePhase()$ function (line 93). In case that $\exists(t, \bullet) \in S_{p_i}$, the function $updatePhase()$ calls the function $upgradePhase()$ (line 95), which transfers $(t, \bullet)$'s phase from 'pre' to 'fin', and 'fin' to 'FIN', but otherwise it does not change $(t, \bullet)$'s phase, e.g., when $p = \text{'pre'}$. Moreover, in case $\nexists(t, \bullet) \in S_{p_i}$, the server at p_i merely adds $(t, \bullet)$ to S_{p_i}. We study each call to $updatePhase()$ in Algorithm 3 and show that it does not remove the currently maximal write record.

- When $(t, \perp, d) : d \in \mathcal{D}$ (lines 76 and 79) arrives at the server at p_i, that server uses the $updatePhase()$ (lines 77 and 80) for making sure that $(t, \bullet, d)$ exists in S_{p_i} (line 93) in a way that can only add a missing record $(t, \perp, d)$ to S_{p_i} (when $(t, \perp, d) \notin S_{p_i}$) or transfer the phase of an existing record in S_{p_i} according to

$upgradePhase()$, which does not remove the currently maximal write record. Moreover, when $(t, \perp, d)$ arrives either from a reader or a writer, p_i updates S_{p_i} in a manner that differs only by the response that p_i sends to the client (lines 78 and 81), i.e., irrelevant to p_i's server state after that send.
– When gossip arrives at p_i (line 82), p_i calculates its new maximal write record in a way that includes both the records in its own storage S_{p_i} and the maximal records reported recently from all servers including itself (lines 83, 85, and 87). Note that p_i might add a new maximal anchor record with tag t whenever it discovers that there is a quorum of servers that have reported a finalized record with tag t (line 87). After calculating these new maximal values, p_i updates its storage S_{p_i} via $updatePhase()$ (lines 84, 86, and 88), in a way that we showed above that p_i does not remove the currently maximal write record.

Parts (1.2) and (1.3). The proofs here follow similar arguments to the ones of Part (1.1); it is even simpler because phs's values are different and thus 'pre' is irrelevant to Part (1.2) and only 'FIN' is relevant to Part (1.3).

We note that the same arguments hold also for any record that has a tag that is higher than these maximal tags.

Lemma 7 (Maximal tags arrive at every server eventually) *Suppose that the server at $p_i \in \mathcal{P}$ calls gossip(T_k) (Sect. 5.1) for an unbounded number of times in Algorithm 3*'s execution, R, such that the triple $T_k := (t_{k,write}, t_{k,read}, t_{k,anchor})$ is the k-th gossip that p_i sends. The server at $p_j \in \mathcal{P}$ receives eventually at least one gossip $(t_{k,write}, t_{k,read}, t_{k,anchor})$, such that each respective tag is not less than its correspondent in $(t_{1,write}, t_{1,read}, t_{1,anchor})$. Moreover, if R is fair, each gossip message arrives within $\mathcal{O}(1)$ asynchronous cycles.

Proof Lemma 6, and Claim 3 facilitate the proof of Claim 4, which implies the first part of this lemma. Claim 3 considers the sequence $t_{k,type}$, where $type \in \{read, write, anchor\}$ and $T_k := (t_{k,write}, t_{k,read}, t_{k,anchor})$.

Claim 3 *The sequence $t_{k,type}$ is non-decreasing.*

Proof The three parts of Lemma 6 show that the server never removes its currently maximal write, read and anchor records. The rest of the proof is implied directly from the fact that line 91 merely calculates the currently maximal write, read and anchor records.

Claim 4 *The server at $p_j \in \mathcal{P}$ receives eventually at least one gossip message $(t_{k,write}, t_{k,read}, t_{k,anchor})$, such that each respective tag is not less than its correspondent in $(t_{1,write}, t_{1,read}, t_{1,anchor})$.*

Proof Let us consider the sequence $t_{k,type}$, where $type \in \{read, write, anchor\}$ and $T_k := (t_{k,write}, t_{k,read}, t_{k,anchor})$. Let $a_k \in R$ be a step in which p_i calls gossip(T_k) for the k-th time in R. Let $a_{depart,k'} \in R$ be the first step that appears after $a_{k'} : k' \in \{1, \ldots\}$ and before $a_{k'+1}$ in R, if there is any such step, in which p_i executes the event of gossip token departure. Let $a_{arrival,k'} \in R$ be the first step that appears after $a_{depart,k'}$ in R, if there is any such step, in which the server at p_j delivers the token that $a_{depart,k'}$

transmits. By the correctness of the gossip functionality (Theorem 1), step $a_{arrival,k'}$ exists eventually. The proof is done, because $T_{k'}$ includes only tags, $t_{k'}$, that are no less than their corresponding elements in T_1 (Claim 3).

We complete this proof by considering the case in which R is fair. Theorem 1, Part (3) implies that step $a_{arrival,k'}$ exists within $\mathcal{O}(1)$ asynchronous cycles. □

Corollary 2 considers the calls to $maxPhase(\mathcal{D})$ (lines 75, 83 and 92), to $maxPhase(\mathcal{D}\backslash\{\text{'pre'}\})$ (lines 74, 85 and 92), and to $maxPhase(\{\text{'FIN'}\})$ (lines 87, and 92). The same arguments as in the proof of Claim 3 imply Corollary 2.

Corollary 2 *Let $phs \in \{\mathcal{D}, \mathcal{D}\backslash\{\text{'pre'}\}, \{\text{'FIN'}\}\}$ and $a_{i,k} \in R$ be a step in which the server at $p_i \in \mathcal{P}$ executes $maxPhase(phs)$ for the k-th time in R. The sequence of $maxPhase_{a_{i,k}}(phs)$'s returned values is non-decreasing.*

8.3 Recovery After the Occurrence of Transient-Faults

The correctness of Algorithm 3 assumes that the system execution is fair. That is, every node participates in the execution within a single asynchronous cycle. This way, the proof bounds the number of asynchronous cycles that it takes the system to remove stale information by receiving the largest tag values and then allowing the system to perform a valid write operation (Definition 7).

Definition 5 (*Notation*) Let π be a (complete) operation in execution R. We use the following notation.

- $\boldsymbol{\hat{Q}(\pi)}$ is the quorum of servers that π's client receives their acknowledgments for π's query.
- Suppose that π is a write operation. $\boldsymbol{Q_{\text{pw}}(\pi)}$ and $\boldsymbol{Q_{\text{fw}}(\pi)}$ are the quorums for π's pre-write, and respectively, finalize phases.
- Let $\boldsymbol{\hat{T}(\pi)}$ be the maximum arriving tag during π's query (line 67). $T(\pi)$ is the tag of π, such that when π is a write operation, $\boldsymbol{T(\pi)} = \hat{T}(\pi) + 1$ is the tag in use during π's pre-write (line 62) and when π is a read operation, $\boldsymbol{T(\pi)} = \hat{T}(\pi)$ is the maximum arriving tag during π's query (line 67).
- $\boldsymbol{c_{start}(R)}$ is the starting state of Algorithm 3's execution R.
- $\boldsymbol{T_{node}(R)}$ is the tag set in the state of any node in $c_{start}(R)$.
- $\boldsymbol{T_{comm}(R)}$ is the tag set in the payload of any message that is delivered during R, but it is never sent during R, because it was in transit in the communication channels in R's starting system state, $c_{start}(R)$.
- $\boldsymbol{T(R)} = T_{node}(R) \cup T_{comm}(R)$ is the set that includes all the tags in $c_{start}(R)$.

For the sake of compatibility of our proposal with the one in [15], we define the set of legal executions (Definition 8) in a way that considers a recovery period from arbitrary (transient) faults, as well as the case in which the system starts from a well-initialized system state (Definition 6).

Definition 6 (*A safe system start*) Let c_{safe} be a system state in which: (1) no client nor server is executing any procedure, (2) the communication channels from the clients to the servers (servers to clients), $pingTx$ and every entry of $pingRx$ (respectively, $pongTx$ and every entry of $pongRx$) include the message $\langle \bot \rangle$ (respectively, $\langle \bot, \bot \rangle$), (3) the communication channels between any two servers, $gossipTx$ and every entry of $gossipRx$ include the message (t_0, t_0, t_0), and (4) the storage S of every server is empty. In this case, we say that c_{safe} is one of the safe system states.

The definition of recovery Algorithm 3's period uses the term a valid client operation (Definition 7).

Definition 7 (*Valid client operations*) Let π be a complete operation in R. Suppose that there exists a system state $c \in R$, such that c appears in R before π's start in $c_{start}(\pi)$ and π's tag is greater than any tag that appears both in c and R's starting system state, i.e., $\max(T(c_{start}(R)) \cap T(c)) < T(\pi)$. In this case, we say that π is *valid*.

Definition 8 specifies legal executions, as such, that follow at least one complete and valid operation.

Definition 8 (*Recovery periods as well as legal executions*) Let $R = R_{recoveryPeriod} \circ R_{legalExecution}$ be an execution of Algorithm 3 that (is legal with respect to the external building blocks in Sect. 5.1 and it) has an arbitrary starting system state, $c_{start}(R)$, with respect to Algorithm 3. Suppose that within a finite number of steps the system reaches a state $c_{start}(\pi_{complete\&valid})$, such that (1) $c_{start}(\pi_{complete\&valid})$ is the starting system state of a complete and valid write operation $\pi_{complete\&valid}$, and (2) R's suffix, $R_{legalExecution}$, starts at $c_{end}(\pi_{complete\&valid})$, where atomicity and liveness hold with respect to any operation that is complete in suffix $R_{legalExecution}$ of R (Definition 3), which starts immediately after $c_{end}(\pi_{complete\&valid})$.

In this case, we refer to $R_{recoveryPeriod}$ and $R_{legalExecution}$ as R's recovery, and respectively, legal periods. We also consider any execution that starts from c_{safe} (Definition 6) to be legal. Namely, $R_{legalExecution}$ is an asynchronous execution of Algorithm 3.

Theorem 2 shows that the system reaches a legal execution eventually. Theorems 3 and 4 show that it takes merely a single complete and valid write operation $\pi_{complete\&valid}$ (Definition 8) to end the recovery period after which the system executes legally, because they demonstrate correct shared-memory emulation. Arora and Gouda [3] refer to the properties demonstrated by Theorem 2 as Convergence and the properties showed by Theorems 3 and 4 as Closure. Our proof shows that fair executions guarantee recovery within $\mathcal{O}(1)$ asynchronous cycles. Once $\pi_{complete\&valid}$ had occurred, the correct system behavior no longer needs the above fairness assumption. Recall that, within $\mathcal{O}(1)$ asynchronous cycles, Algorithm 3's execution reaches a suffix in which the correctness of gossip and quorum-based communication is guaranteed (Theorem 1). Therefore, Theorem 2 considers an execution of Algorithm 3 that is legal with respect to the external building blocks in Sect. 5.1, because it demonstrates that the system reaches suffix $R_{no\ incomplete}$ within $\mathcal{O}(1)$ asynchronous cycles.

Theorem 2 (Recovery after the occurrence of transient-faults) *Let R be a fair execution of Algorithm* 3 that (is legal with respect to the external building blocks in Sect. 5.1

and it) has an arbitrary starting system state, $c_{start}(R)$, with respect to Algorithm 3. Within $\mathcal{O}(1)$ asynchronous cycles, execution $R = R' \circ R_{no\ incomplete}$ has a suffix $R_{no\ incomplete}$ that does not include incomplete operations. Moreover, within $\mathcal{O}(1)$ asynchronous cycles, execution $R_{no\ incomplete}$ reaches a suffix, $R_{completeNonStable}$, that does not include invalid operations.

Proof The proof is implied by Claim 7, which uses claims 5 and 6 . Leveraging Theorem 1, Claim 5 shows that Algorithm 3's executions stop having incomplete operations.

Claim 5 *Let R be a fair execution of Algorithm* 3 in which the gossip functionality behaves correctly. Within $\mathcal{O}(1)$ asynchronous cycles, R includes a suffix, $R_{no\ incomplete}$, that does not include: (1) operations that are incomplete in R, nor (2) incomplete client requests or server replies in R. Moreover, (3) $\exists c \in R_{no\ incomplete} : \forall p_i \in \mathcal{P} : \exists t \in \mathcal{T} : t' \in (T(R_{no\ incomplete})) \implies \exists (t, \bullet) \in S_{p_i} : t \geq t'$ in c.

Proof **Part (1).** Lemma 5 implies that all incomplete operations end within $\mathcal{O}(1)$ asynchronous cycles.
Part (2). Suppose that all operations in $R_{no\ incomplete}$ are complete, i.e., no incomplete request or replies enter the system throughout $R_{no\ incomplete}$. (Due to Part (1) of this proof, we can make this assumption without losing generality.) Lemma 5 implies the correct behavior of the quorum-based communication functionality within $\mathcal{O}(1)$ asynchronous cycles, which implies Part (2).

For the sake of simple presentation, the rest of this proof assumes that throughout $R_{no\ incomplete}$, all of client requests and server replies were indeed (Sect. 5.1).
Part (3). We start by showing that $T_{comm}(R_{no\ incomplete}) = \emptyset$ (Definition 5). Parts (1) and (2) of this proof says that $R_{no\ incomplete}$ does not include the delivery of messages that were never sent in R. This implies $T_{comm}(R_{no\ incomplete}) = \emptyset$, because $T_{comm}()$'s definition considers any message that is delivered but never sent during (since they were in transit at the starting system state of $R_{no\ incomplete}$).

Due to the above, we only show that within $\mathcal{O}(1)$ asynchronous cycles in $R_{no\ incomplete}$, the system reaches a system state $c \in R_{no\ incomplete}$, such that $\forall p_i \in \mathcal{P} : \exists t \in \mathcal{T} : t' \in (T_{node}(R_{no\ incomplete})) \implies \exists (t, \bullet) \in S_{p_i} : t \geq t'$. Let $t' \in (T_{node}(R_{no\ incomplete}))$. Suppose that t' appears in the client state at node $p_j \in \mathcal{P}$. By the assumption that this theorem makes about R fairness, we know that p_j's client operation terminates within $\mathcal{O}(1)$ asynchronous cycles (Lemma 5). Once that happens, the client state no longer includes any tag value (cf. part (c) of the quorum-based communication service and Theorem 1). Suppose that t' is part of the server state, i.e., $\exists p_j \in \mathcal{P} : (t', \bullet) \in S_{p_j}$. Let us consider a choice of p_j and t', such that t' is maximal. By Lemma 7, within $\mathcal{O}(1)$ asynchronous cycles, p_i's server receives at least one gossip that includes a tag $t'' \geq t'$ that is not smaller than t'. The proof is done by replacing t with t'' in the invariant that we need to prove, i.e., $\forall p_i \in \mathcal{P} : \exists t'' \in \mathcal{T} : t' \in (T_{node}(R_{no\ incomplete})) \implies \exists (t'', \bullet) \in S_{p_i} : t'' \geq t'$.

Claim 6 shows that a $R_{no\ incomplete}$'s operation, π, uses a tag that is not smaller than any (maximal) tag $T_{maxQuery}(\pi)$ on the servers that participate in π's query quorum, where $tags(C, D) = \{t : (t, \bullet, d) \in S_{p_j} \wedge d \in D \wedge p_j \in C\}$ (Definition 4) and $T_{maxQuery}(\pi) = \max tags(\hat{Q}(\pi), \mathcal{D} \backslash \{\text{'pre'}\})$ in $c_{start}(\pi) \in R_{no\ incomplete}$.

Claim 6 *Let π be an $R_{no\ incomplete}$'s operation. $T(\pi) \geq T_{maxQuery}(\pi)$ in the system state $c_{start}(\pi) \in R_{no\ incomplete}$. Moreover, $T(\pi) > T_{maxQuery}(\pi)$ when π is a write operation.*

Proof Due to the correctness of the quorum-based communication functionality during $R_{no\ incomplete}$ (Claim 5), Corollary 2, as well as lines 61, 67, and 73 to 75, it holds that $\hat{T}(\pi)$ is not smaller than any write or read tag in $S_{p_j} : p_j \in \hat{Q}(\pi)$ in $c_{start}(\pi)$. Moreover, $T(\pi) \geq \hat{T}(\pi)$ (Definition 5) and $T(\pi) > \hat{T}(\pi)$ when π is a write operation. Thus, in $c_{start}(\pi)$, it holds that $T(\pi)$ is not smaller than any tag in $S_{p_j} : p_j \in \hat{Q}(\pi)$ (and it is actually greater when π is a write operation).

Claim 7 implies that any write operation π_{write} in $R_{completeNonStable}$ is valid with respect to R and by that we complete the proof.

Claim 7 *Within $\mathcal{O}(1)$ asynchronous cycles, execution $R_{no\ incomplete}$ reaches a suffix, which we denote by $R_{completeNonStable}$, such that for any write operations, π_{write}, in $R_{completeNonStable}$, it holds that in $c \in R_{completeNonStable}$ we have that $T(\pi_{write}) > \max(T(R_{no\ incomplete}))$ holds.*

Proof The proof is implied by Part (3) of Claim 5 and Claim 6.

□

8.4 Atomicity of Algorithm 3

We demonstrate that, after a recovery period (Definition 8), Algorithm 3 emulates shared atomic read/write memory. Some elements of the following proof are similar to arguments in [15, Theorem 1]. Note that Theorem 3 considers $R_{legalExecution}$ but does not require fairness. By that, it merely assumes that at least a single complete and valid write operation occurred during the recovery period (Definition 8) or that the system starts in a safe state (Definition 6).

Theorem 3 (Atomicity) *Algorithm* 3 is atomic in $R_{legalExecution}$.

The $\prec$ order satisfies the sufficient conditions for atomicity (Corollary 3), which we borrow from [15].

Corollary 3 (Lemma 2 in [15]) *Let Π be the set of all operations in R. Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π that satisfies: (1) when π_1's return precedes π_2's start in R, $\pi_2 \prec \pi_1$ is false. (2) When $\pi_1 \in \Pi$ is a write operation and $\pi_2 \in \Pi$ is any client operation, either $\pi_1 \prec \pi_2$ or $\pi_2 \prec \pi_1$ holds (but not both). (3) The value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or v_0, which is the default object value in the absence of such write).*

Definition 9 Define $\pi_1 \prec \pi_2$ if (i) $T(\pi_1) < T(\pi_2)$, or (ii) $T(\pi_1) = T(\pi_2)$, π_1 is a write and π_2 is a read.

We show that $\prec$ satisfies the conditions of Corollary 3. The proof of the closure property follows similar arguments to the ones made by Cadambe et al. [15]. It shows that by the time operation π ends, the tag $T(\pi)$ has finished propagating and installing the messages $\langle T(\pi), \bullet, \text{'fin'}\rangle$ in the storage of at least one quorum of servers (Lemma 8). It uses the visibility of $T(\pi)$ for claiming that the query phase of any operation that starts after π's end, retrieves a tag that is at least as large as $T(\pi)$ (Lemma 9). This is the basis of showing that each write operation has a unique tag (Lemma 10). We complete the proof of Theorem 3 by demonstrating conditions (1) and (2) of Corollary 3 (using Lemmas 9 and 10), as well as condition (3) by considering read and write operations (Algorithm 3) during $R_{legalExecution}$.

Lemma 8 is a variation on Lemma 3 in [15]. We use Lemma 6 for arguing that the servers (Algorithm 3) store the currently maximal records (and any record with a higher tag). This variation is needed, because Lemma 8 considers only operations that start after the (last) valid and complete write operation $\pi_{complete\&valid}$ (or a system that starts in a safe system state, cf. Definition 6).

Lemma 8 (Storing the operation records) *Suppose that π is a complete (read or write) operation in $R_{legalExecution}$. There is a quorum $Q_{fw}(\pi) \in \mathcal{Q}$, such that all of its servers store the triple $(t, w, \text{'fin'})$, where $t = T(\pi)$ and $w \in \mathcal{W} \cup \{\bot\}$ in $c_{end}(\pi)$ and in every system state after $c_{end}(\pi)$.*

Proof Let $Q_{\text{fw}}(\pi)$ the quorum that π's client (at node p_i) receives responses from $Q_{\text{fw}}(\pi)$'s servers during π's finalize phase (lines 63 and 68). Since π is complete, as well as the functionalities of gossip and quorum-based communication are correct in $R_{legalExecution}$ (Theorem 1), it is true that the server at node $p_j \in Q_{\text{fw}}(\pi)$ responds to π's finalize message (line 79) at some step $a_{\text{fw},j} \in R_{legalExecution}$. Note that: (i) p_j's response arrives eventually to p_i's writer and that occurs before the system reaches $c_{end}(\pi)$, because $p_j \in Q_{\text{fw}}(\pi)$, as well as (ii) the servers (Algorithm 3) keep in their storage the currently maximal (write, read, and anchor) records and any received record with a tag that is higher than the ones in these records (Lemma 6).

Lemma 9 (Similar to Lemma 4 in [15]) *Let $\pi_i : i \in \{1, 2\}$ be two complete operations in R, such that each π_i starts immediately after the system states $c^s_i \in R : i \in \{1, 2\}$ and returns immediately before $c^r_i \in R : i \in \{1, 2\}$. Assume that c^r_1 appears before c^s_2 in R. (1) $T(\pi_2) \geq T(\pi_1)$ and (2) when π_2 is a write operation, $T(\pi_2) > T(\pi_1)$.*

Proof Let $\hat{T}(\pi)$ be the maximum arriving tag during π's query (lines 61 and 67). It suffices to show that $\hat{T}(\pi_2) \geq T(\pi_1)$ (Claim 8), because when π_2 is a read, $T(\pi_2) = \hat{T}(\pi_2)$, and when π_2 is a write, $T(\pi_2) > \hat{T}(\pi_2)$ (see the pseudo-code of the reader and writer in Algorithm 3).

Claim 8 $\mathbf{\hat{T}}(\pi_\mathbf{2}) \geq \mathbf{T}(\pi_\mathbf{1})$.

Proof Let $\hat{Q}(\pi_i)$ be the set of nodes that their servers respond to π_i's query (lines 61 and 67). Note the existence of node $p_j \in \hat{Q}(\pi_2) \cap Q_{\text{fw}}(\pi_1)$ (Lemma 2) that its server responds to π_2's query with $(t, \bullet, \text{'qry'})$ (line 73) immediately after some system state $\hat{c}_{2,j} \in R$, where t is the highest tag of a finalized (or FINALIZED) record that p_j stores in S_{p_j}. We argue that $t \geq T(\pi_1)$, because $(T(\pi_1), \bullet, d) \in S_{p_j} : d \in \mathcal{D}\backslash\{\text{'pre'}\}$ in $\hat{c}_{2,j}$ and t is S_{p_j}'s highest finalized tag in $\hat{c}_{2,j}$. In detail, we argue the following.

1. The fact that $p_j \in Q_{\text{fw}}(\pi_1)$ implies $(T(\pi_1), \bullet, d) \in S_{p_j} : d \in \mathcal{D}\setminus\{\text{‘pre’}\}$ as long as $\hat{c}_{2,j}$ appears after $c_{end}(\pi_1)$ in R (Lemma 8). Moreover, $\hat{c}_{2,j}$ indeed appears after $c_{end}(\pi_1)$ in R, since c_1^r appears before c_2^s in R (by this lemma assumption) and c_2^s cannot appear after $\hat{c}_{2,j}$ (by the fact that $\hat{c}_{2,j}$ appears immediately before the response to a query that is sent immediately after c_2^s).
2. The fact that $p_j \in \hat{Q}(\pi_2)$ implies that p_j responds to π_2's query (by $\hat{Q}(\pi_2)$'s definition),
3. The server at p_j replies with $(t, \bullet, \text{‘qry’})$ to π_2's query, such that $(t, \bullet, d) \in S_{p_j} : d \in \mathcal{D}\setminus\{\text{‘pre’}\}$, where t is S_{p_j}'s highest finalized (or FINALIZED) tag (line 73) in $\hat{c}_{2,j}$.

Since $t \geq T(\pi_1)$, it holds that π_2's query phase includes the reception of a response with a tag that is no smaller than $T(\pi_1)$. Thus, $\hat{T}(\pi_2) \geq T(\pi_1)$.

□

Lemma 10 (cf. Lemma 5 in [15]) *Let π_1 and π_2 be two write operations in R. $T(\pi_1) \neq T(\pi_2)$.*

Proof Denote by $id_i :\in \{1, 2\}$ the identifier of the node that invokes operation π_i.

Note that $id_1 \neq id_2$ implies $T(\pi_1) \neq T(\pi_2)$, because $T(\pi_i) = (z_i, id_i)$ (lines 62, 63 and 64 , Algorithm 3).

Thus, until the end of this proof, we focus on the case in which $id_1 = id_2$. The client (at node p_i) performs sequentially the operations π_1 and π_2 (Sect. 2.2), i.e., one of them ends before the other starts. Let us assume, without loss of generality, that π_1 ends before π_2 starts. $T(\pi_2) > T(\pi_1)$ (Lemma 9) implies that $T(\pi_2) \neq T(\pi_1)$. □

Proof of Theorem 3 For any two operations π_1, π_2, the definition of $\prec$ (Corollary 3) says $\pi_1 \prec \pi_2$ when: (i) $T(\pi_1) < T(\pi_2)$, or (ii) $T(\pi_1) = T(\pi_2)$ as long as π_1 is a write and π_2 is a read. Suppose that operations π_1 and π_2 occur in Algorithm 3's legal execution $R_{legalExecution}$. After verifying that $\prec$ is indeed a partial order, we show the three properties of Corollary 3.

The relation $\prec$ is a partial order. We demonstrate that $\pi_1 \prec \pi_2 \implies \pi_2 \not\prec \pi_1$ by assuming that this statement is false, i.e., $\pi_1 \prec \pi_2 \wedge \pi_2 \prec \pi_1$, and then show a contradiction. Note that $(T(\pi_1) \leq T(\pi_2)) \wedge (T(\pi_2) \leq T(\pi_1)) \implies T(\pi_1) = T(\pi_2)$ ($\leq$'s definition). Therefore, π_1 is a write and π_2 is a read (Part (ii), Definition 9). Using symmetrical arguments, π_2 is a write and π_1 is a read. A contradiction.

Property (1) of Corollary 3.

Assume that π_1 returns before π_2 starts in R.

We show that whether π_2 is a read or a write, it holds that $\pi_2 \prec \pi_1$ is false.

– When π_2 is a read, $T(\pi_2) \geq T(\pi_1)$ (Lemma 9, as well as the assumption that π_1 returns before π_2 starts). Thus, $\pi_2 \prec \pi_1$ is false, because otherwise, by Definition 9 of the order $\prec$, it holds that: (i) $T(\pi_1) > T(\pi_2)$, which contradicts the above, or (ii) π_2 is a write (Definition 9 of the order $\prec$). Moreover, with respect to case (ii), if π_2 is a write, $T(\pi_2) > T(\pi_1)$ (Lemma 9, as well as the assumption that π_1 returns before π_2 starts). Thus, $\pi_1 \prec \pi_2$ is true (case (i), Definition 9 of the order $\prec$). Moreover, $\pi_2 \prec \pi_1$ is false ($\prec$ is a partial order).

– When π_2 is a write $T(\pi_2) > T(\pi_1)$ (Lemma 9, as well as the assumption that π_1 returns before π_2 starts). Thus, $\pi_1 \prec \pi_2$ is true (case (i), Definition 9 of the order $\prec$). Moreover, $\pi_2 \prec \pi_1$ is false ($\prec$ is a partial order)

Property (2) of Corollary 3.

Lemma 10 implies that only case (i) of Definition 9 holds. This implies Property (2), i.e., either $\pi_1 \prec \pi_2$ or $\pi_2 \prec \pi_1$ (but not both) hold.

Property (3) of Corollary 3.

We show that every read operation π in a legal execution $R_{legalExecution}$ returns a value that a preceding, according to $\prec$, write operation writes. (In the absence of such write operations, the read operation π returns v_0, which is the default object value, line 71).) To that end, we argue that: (i) there is a unique coupling between object version values and tag values and (ii) the read operation π returns the value associated with $T(\pi)$.

(i) Unique coupling between object version values and tag values.

Recall that the system reaches $R_{legalExecution}$ after the system has performed at least one complete and valid write operation $\pi_{greatFIN} \in R_{completeNonStable}$ (Definition 8 and Theorem 2).

After $\pi_{greatFIN}$, any succeeding write operation $\pi_{furtherWrite}$ in $R_{legalExecution}$ couples uniquely between versions of the data object and write operations in R (Sect. 2.2). We know that all written versions are uniquely associated with tag values (Lemma 10). We note that even when starting the system in a state that includes no written object values, the servers reply with $(t_0, w_{0,i}, \text{'fin'})$ (line 71) and the reader returns the decoding of that value (line 69).

(ii) The read operation π returns the value associated with $T(\pi)$.

The complete read operation $\pi_{legitimateRead} \in R_{legalExecution}$ returns a value that is the result of retrieving and inverting the MDS code Φ using k coded elements (line 69 and Definition 4). These k coded elements were obtained at some previous point by applying Φ to the value associated with $T(\pi)$, where $\pi \in \{\pi_{greatFIN}, \pi_{furtherWrite}\}$ (line 62). Therefore, the read operation π returns the value associated with $T(\pi)$ due to the correctness of Φ (Sect. 2.2). □

8.5 Liveness of Algorithm 3

Definition 10 (*Liveness criteria*) Suppose that there are no more than f crashed severs, e data-corrupting malicious server, and that $k_{threshold} \in \{1 \ldots, N - 2(f + e)\}$. In any fair and legal execution of Algorithm 3, it holds that: (1) every operation terminates, and (2) the servers replying to a reader's finalize phase includes at least $k_{threshold}$ (different) coded elements (and thus read operations can decode the retrieved values).

Theorem 4 (Liveness) *The liveness criteria (Definition 10) hold in Algorithm 3*'s fair and legal executions.

Proof Note that Lemma 5 implies Part (1) of the liveness criteria (Definition 10). Therefore, we focus on proving that read operations can decode the retrieved values (Part (2) of Definition 10). I.e., at least k servers include coded elements in their replies to a reader's finalize phase. The proof is implied from Claims 9 and 10 and the fact that Algorithm 3's servers do not remove records from their storage.

Claim 9 *The query of a read operation π_r in $R_{legalExecution}$ always returns a tag t that is either t_0 or refers to the tag of a write operation π_w that had a complete pre-write phase in R.*

Proof Definition 8 implies that π_w always occurred before the legal execution (or the servers only consider the default tuple with the tag t_0). Lemma 6 says that the servers do not remove their maximal records. Upon the arrival of π_r's query message, the server replies with π_w's tag (line 74), which is t.

Claim 10 *As long the no server removes the record $(t, \bullet)$ from its storage, if it had any such record in $c_{start}(\pi_r)$, at least k servers include coded elements in their replies to π_r's finalize phase.*

Proof Let $Q_{\text{pw}}(t)$ denote the set of nodes that their servers acknowledge the pre-write phase of the write operation π_w for which $t = T(\pi_w)$. Let c_i be the system state that occurs immediately before the server at p_i acknowledges π_r's finalize message (line 81). We show that the storage S_{p_i} of every node $p_i \in Q_{\text{pw}}(t) \cap Q_{\text{fw}}(t)$ includes a coded element in c_i. Since $p_i \in Q_{\text{pw}}(t)$, it holds that $(t, w_{t,i}, \bullet) \in S_{p_i}$ in any system state that follows the step in which p_i received π_w's pre-write message (line 76 and by the assumption of this claim that no server removes the record $(t, \bullet)$ from its storage). Note that $p_i \in Q_{\text{fw}}(t)$ indeed acknowledges the reader's finalize message, because of Claim 9 and the fact that c_i appears in R after p_i acknowledges that pre-write message. Therefore, p_i includes in its reply the coded element $w_{t,i}$. By the correctness of the quorum-based communication during legal executions (Theorem 2, Claim 5), π_r receives at least k coded elements in its finalize phase, because $|Q_{\text{pw}}(t) \cap Q_{\text{fw}}(t)| \geq k$ (Part(1) of Lemma 2).

□

9 A Bounded Set of Relevant Server Records

Algorithm 3's servers store the entire set of records that have arrived from the clients and the gossip service. This is in addition to the records that originated from the system starting state. To the end of bounding the number records that each server needs to store, we consider the relevance of a record with respect to the way that the servers use it after any point of time, i.e., a record is irrelevant in system state $c \in R_{legalExecution}$ if the server at $p_i \in \mathcal{P}$ never use it after c for responding to a client request. Theorem 5 and Corollary 4 point out a set that includes all relevant records and bound it by $N + \delta + 3$ during executions $R_{legalExecution}$ in which there are no more than δ write operations that occur concurrently with any read operation.

Definition 11 (*Tag visibility*) Let R be an execution of Algorithm 3, π_r be a read operation and π_w be a write operation in R. Denote by $c_{visibility}(\pi_r) = c_{end}(\pi_r)$, which refers to π_r's ending system state. We say that π_r has visibility in R starting from $c_{visibility}(\pi_r)$. Moreover, denote by $c_{visibility}(\pi_w) \in R$ either:

(i) the first system state, if such a state exists, for which a quorum $Q \in \mathcal{Q}$ of non-failing nodes that their servers store the finalized record $(T(\pi_w), \bullet, d) \in S_{p_j \in Q} : d \in \mathcal{D} \setminus \{\text{‘pre’}\}$, or

(ii) when case (i) does not hold in R (because operation π_w fails in R), $c_{visibility}(\pi_w) = c_{end}(\pi_w)$, which refers to π_w's ending system state. When case (i) holds for π_w, we say that π_w has visibility in R starting from $c_{visibility}(\pi_w)$. Otherwise, π_w's visibility is not guaranteed in R.

Definition 12 (*Explicit and implicit FINALIZED tags and records*) Suppose that the server at node p_i stores a finalized (or FINALIZED) record $r = (t, \bullet, d) \in S_{p_i} : t \in \mathcal{T} \wedge d \in \mathcal{D} \setminus \{\text{‘pre’}\}$ in system state $c \in R$. In this case, we say that tag t and record r are explicitly finalized (with respect to the server) at p_i. Moreover, we say that tag t and record r are explicitly FINALIZED at p_i when $(t, \bullet, \text{‘FIN’}) \in S_{p_i} : t \in \mathcal{T}$ in system state $c \in R$.

Suppose that the server at node p_i stores two records $r_1, r_2 \in S_{p_i}$, such that $\exists_{p_j \in \mathcal{P}} \forall_{x \in \{1,2\}} t_x = (z_x, j) \wedge r_x = (t_x, \bullet)$ in system state $c \in R$ that their tags, $t_x = (z_1, j)$, and respectively, $t_x = (z_2, j)$, are associated with the client at p_j. Moreover, suppose that $t_1 < t_2$. In this case, we say that tag t_1 and record r_1 are implicitly FINALIZED (in with respect to the server) at p_i. We denote S_{p_i}'s explicit FINALIZED records in c by $S_{i,\text{expFIN}} := \{(t, \bullet, \text{‘FIN’}) \in S_{p_i}\}$ and S_{p_i}'s implicitly FINALIZED records in c by $S_{i,\text{impFIN}} := \{((z_1, j), \bullet) \in S_{p_i} : \exists ((z_2, j), \bullet) \in S_{p_i} \wedge z_1 < z_2\}$.

Claim 11 shows that an implicitly FINALIZED record at a server implies explicitly FINALIZED records at a server quorum.

Claim 11 *Suppose that $R_{legalExecution}$ includes a write operation π, such that in system state $c \in R_{legalExecution}$ it holds that $T(\pi)$ is implicitly FINALIZED at p_i. (1) π is visible in c. (2) There is a quorum $Q \in \mathcal{Q}$ of nodes that their servers store the FINALIZED record $(T(\pi), \bullet, \text{‘FIN’}) \in S_{p_j \in Q}$. Suppose that in c it holds that $T(\pi)$ is explicitly FINALIZED at p_i, i.e., $(T(\pi), \bullet, \text{‘FIN’}) \in S_{p_i}$. (3) π is visible in c.*

Proof We start the proof by showing that π includes the entire execution of the FINALIZED phase before $R_{legalExecution}$ reaches the system state c. We do that by demonstrating that π is not an incomplete operation, nor a failed one. Recall that Claim 5 implies that $R_{legalExecution}$ does not include (write) operations that are incomplete and thus π is not an incomplete operation. This claim assumes that in system state c, it holds that $T(\pi)$ is implicitly FINALIZED at p_i. This means that, in c, the server at node p_i stores two records $r_1, r_2 \in S_{p_i} : r_x = (t_x, \bullet), t_x \in \mathcal{T} \wedge t_x = (z_x, j) \wedge p_j \in \mathcal{P}$, such that $T(\pi) = t_1 < t_2$ (Definition 12). By the assumption that each node $p_j \in \mathcal{P}$ lets its client to run just one procedure at a time, by the assumption that failing clients do not resume (Sect. 2.2.3), and by the writer code (lines 61 and 65), we have that π is not a failed operation. Therefore, π is a complete write operation that ends before c. In particular, π's finalized and FINALIZED phases are done before R reaches c and thus parts (1) and (2) are correct (by Definition 11 and the correct operation of the quorum-based communications Theorem 1). To show that part (3) also holds, we note that during $R_{legalExecution}$, any write operation π, which is after $\pi_{complete\&valid}$, updates to the record $(T(\pi), \bullet, \text{‘FIN’}) \in S_{p_i}$ occurs only after the completion of the finalized phase (line 63 and 64). Thus, visibility is implied (Definition 11).

Definition 13 (*The done system state $c_{done}(\pi)$*) Let R be an execution of Algorithm 3 and π be a client (read or write) operation in R. Let $a_k(\pi) \in R$ be the step in which a server (at node $p_i \in \mathcal{P}$) adds or updates the record $(T(\pi), \bullet)$ to its server storage, S_{p_i} for the k-th time. This update could be due to the π operation itself, another read operation $\pi_r \neq \pi$ for which $T(\pi_r) = T(\pi)$, or the arrival of a gossip message $(\bullet, T(\pi), \bullet)$. Denote $c_0(\pi) := c_{start}(\pi)$, $c_k(\pi)$ is the system state that immediately follows $a_{i,k}(\pi)$ and $c_{last}(\pi) = c_\ell(\pi)$, where ℓ is the maximum value of ℓ for which $\exists c_\ell(\pi) \in R$. We denote by $c_{done}(\pi) \in \{c_{last}(\pi), c_{end}(\pi)\}$ the system state that appears latest in R between $c_{last}(\pi)$ and $c_{end}(\pi)$.

Definition 14 (*Concurrent operations*) Let π_1 and π_2 be two operations in R. Suppose that $\nexists x, y \in \{1, 2\} : x \neq y$, such that $c_{done}(\pi_x)$ appears before $c_{start}(\pi_y)$ in R. In this case, we say that π_1 and π_2 appear to be concurrent in R.

We note that one way to explain Definition 14, is to say the following. When $c_{done}(\pi_x)$ appears before $c_{start}(\pi_y)$ in R, we can say that R orders π_x before π_y sequentially. Moreover, π_1 and π_2 appears to be concurrent in R if, and only if, R neither orders π_x before π_y nor π_y before π_x.

Definition 15 (*δ-bounded concurrent write operations during any read in R*) Suppose that for every read operation π_r in R, it holds that there are at most δ write operations in R that are concurrent with π_r. In this case, we say that the number of concurrent write operations that occur in R during any read operation is bounded by δ in R.

Definition 16 (*Record relevance*) Let $r = (t, \bullet) \in S_{p_i} : t \in \mathcal{T}$ be a record that the server at node $p_i \in \mathcal{P}$ stores in system state $c \in R$. Suppose that there is a step a_i that appears in R after c and in which the server at node p_i responses to a (1) writer query request (line 75), (2) reader query request (line 74) or (3) reader finalized request (lines 79) with a message that includes tag $t' \leq t$. In this case, we say that tag t and record r are of relevance to c with respect to a (1) writer query request, (2) reader query request, and respectively, (3) reader finalized request.

Definition 17 (*The $T_{i,writeQuery}$, $T_{i,readQuery}$ and $T_{i,readFinalized}$ sets*) Let $p_i \in \mathcal{P}$ be a node with a server. Let $t_{i,FINALIZED} = t_{i,1}, t_{i,2}, \ldots : (t_{i,k}, \bullet) \in (S_{i,\text{expFIN}} \cup S_{i,\text{impFIN}})$ (Definition 16) be a (possibly empty) sequence tags in a descending order that are explicitly or implicitly FINALIZED at p_i in system state c. Let $maxT_{i,FINALIZED} = \max\{t_{i,x} \in t_{i,FINALIZED}\}$ and $T_{i,FINALIZED} = \{t_{i,x} \in t_{i,FINALIZED} : x \leq \delta + 1\}$. Let $T_{i,notYetFIN} = \{t : (t, \bullet) \in S_{p_i} \setminus (S_{i,\text{expFIN}} \cup S_{i,\text{impFIN}})\}$ be a (possibly empty) set of tags that are at p_i's record storage and are not in $T_{i,FINALIZED}$ in system state c. Let $T_{i,writeQuery} = \{\max\{t : (t, \bullet) \in S_{p_i}\}\}$, $T_{i,readQuery} = \{\max\{t : (t, \bullet, d) \in S_{p_i} : d \in (\mathcal{D} \setminus \{\text{'pre'}\})\}\}$ and $T_{i,readFinalized} = T_{i,notYetFIN} \cup T_{i,FINALIZED}$ in system state c.

Lemma 11 *Suppose that during any read operation in $R_{legalExecution}$ there are at most δ concurrent write operations. Suppose that $R_{legalExecution}$ includes a read operation π_r and a step $a_i \in R_{legalExecution}$ in which the server at p_i responds with $(T(\pi_r), \bullet)$ to π_r's finalize request (line 79), such that $R_{legalExecution}$ includes a write operation π_w for which $T(\pi_w) = T(\pi_r)$. (If there is more than just one such operation, we select* *the latest one that appears before π_r and note that by Theorem 3 these operations cannot be concurrent.)*

It holds that $T(\pi_r) \in T_{i,readFinalized}$ in any system state $c \in R_{legalExecution}$ that is between $c_{i,in} \in R_{legalExecution}$ and $c_{i,out} \in R_{legalExecution}$, where $c_{i,in}$ is $R_{legalExecution}$'s first system state for which $(T(\pi_r), \bullet) \in S_{p_i}$ holds and $c_{i,out}$ is the system state that immediately precedes a_i.

Proof The proof is implied by the following claims.

Claim 12 *Let $c \in R_{legalExecution}$ be a system state. $|S_{i,notYetFIN}| \leq N$ holds in c (Definition 17).*

Proof By the definition of $S_{i,\text{impFIN}} := \{((z_1, j), \bullet) \in S_{p_i} : \exists((z_2, j), \bullet) \in S_{p_i} \wedge z_1 < z_2\}$ (Definition 12), it holds that $S_{i,notYetFIN} := S \setminus (S_{i,\text{expFIN}} \cup S_{i,\text{impFIN}})$ does not include any record $((z_1, j), \bullet)$ for which $((z_2, j), \bullet) \in S_{i,notYetFIN}$ and $z_1 < z_2$. Therefore, every client can have at most one tag that appear in a record that belongs to $S_{i,notYetFIN}$. The proof of this claim is implied by the upper bound on the number of clients, which is N (Sect. 2.1.1).

Claim 13 *Let t_{start} be the maximum visible tag in $c_{start}(\pi_r) \in R$. It holds that $t_{start} \leq T(\pi_r)$.*

Proof By the assumption that t_{start} is the maximum visible tag in $c_{start}(\pi_r)$, it holds that there is a quorum $Q \in \mathcal{Q}$ of nodes that their servers store the finalized record $(T(\pi), \bullet, d) \in S_{p_{j \in Q}} : d \in \mathcal{D} \setminus \{\text{‘pre’}\}$ (Definition 11), such that π is a write operation in R and $T(\pi) = t_{start}$. Let $\hat{Q}(\pi_r)$ be the set of nodes that π_r's client receives their query responses (Definition 5). Note the existence of node $p_j \in \hat{Q}(\pi) \cap Q$ (Lemma 2) that its server responds to π_r's query with a tag that is at least t_{start} (line 81). The rest of the proof is implied by line 67 and Part (2) of Theorem 1.

Claim 14 *Let $t_{i,start} := \max T_{i,FINALIZED}$ be $T_{i,FINALIZED}$'s the maximum tag in $c_{start}(\pi_r) \in R$. It holds that $t_{i,start} \leq T(\pi_r)$.*

Proof Part (1) of Claim 11 implies that tag $t_{i,start}$ is visible in $c_{start}(\pi_r)$. Let t_{start} be the maximal tag that has visibility in $c_{start}(\pi_r) \in R$, i.e., $t_{i,start} \leq t_{start}$. By Claim 13, we have that $t_{i,start} \leq t_{start} \leq T(\pi_r)$, which implies this claim. □

Claim 15 *Let $t_{visibility}(c) \in \mathcal{T}$ be the maximum explicitly visible tag in system state $c \in R_{legalExecution}$.*

Suppose that $T(\pi_r) \geq t_{visibility}(c)$ in system state $c \in R_{legalExecution}$ that is between $c_{j,in} \in R_{legalExecution}$ and $c_{j,out} \in R_{legalExecution}$, where $p_j \in \mathcal{P}$. In c, it holds that:

1. $T(\pi_r) \geq maxT_{i,FINALIZED}$, *and*
2. $(T(\pi_r), \bullet) \in S_{p_j}$ *implies* $T(\pi_r) \in \{maxT_{j,FINALIZED}\} \cup \{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\}$.

Proof **Part (1).** Recall that $maxT_{j,FINALIZED} = \max\{t_{j,x} \in t_{j,FINALIZED}\}$ (Definition 17), where $t_{j,FINALIZED} = t_{j,1}, t_{j,2}, \ldots : (t_{j,k}, \bullet) \in (S_{j,\text{impFIN}} \cup S_{j,\text{expFIN}})$

(Definition 16). Let us consider any tag that is either in $S_{j,\text{impFIN}}$ or $S_{j,\text{expFIN}}$, i.e., any tag that is FINALIZED either (i) implicitly or (ii) explicitly. That is, we look at the cases in which (i) $(T(\pi_r), \bullet) \in S_{p_j} : p_k \in \mathcal{P} \wedge T(\pi_r) = (z_1, k) \wedge \exists((z_2, k), \bullet) \in S_{p_j} : z_1 < z_2$ in c, or (ii) $(T(\pi_r), \bullet, \text{'FIN'}) \in S_{p_j}$ in c. Parts (1), and respectively, (3) of Claim 11 imply that $T(\pi_r)$ has visibility in c. This claim assumption says that $T(\pi_r) \geq t_{visibility}(c)$. Therefore, $T(\pi_r) \geq maxT_{j,FINALIZED}$ (Claim 14).
Part (2). By Definitions 12 and 17 , $(T(\pi_r), \bullet) \in S_{p_j}$ implies that either $(T(\pi_r), \bullet) \in (S_{j,\text{impFIN}} \cup S_{j,\text{expFIN}})$ or $(T(\pi_r), \bullet) \in T_{j,notYetFIN}$. Part (1) of this proof consider the former case and implies that $T(\pi_r) \in \{maxT_{j,FINALIZED}\}$ in c. The latter refers to the cases that Part (1) of this proof do not consider. That is, $(T(\pi_r), \bullet, d) \in S_{p_j} : d \in \mathcal{D} \backslash \{\text{'FIN'}\} \wedge p_k \in \mathcal{P} \wedge T(\pi_r) = (z_1, k) \wedge \nexists((z_2, k), \bullet) \in S_{p_j} : z_1 < z_2$, which implies $(T(\pi_r), \bullet, d) \in T_{j,notYetFIN}$. □

Claim 16 *Let $t_{visibility}(c) \in \mathcal{T}$ be the maximum explicitly visible tag in system state $c \in R_{legalExecution}$ and $c_1, c_2, \ldots$ be a sequence of all system states in $R_{legalExecution}$ (in the order that they appear in $R_{legalExecution}$).*

1. *$(t_{visibility}(c), \bullet) \in S_{p_j}$ implies $t_{visibility}(c) \in \{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\} \cup \{maxT_{j,FINALIZED}\}$ in c.*
2. *The sequence $t_{visibility}(c_1), t_{visibility}(c_2), \ldots$ is monotonically increasing, i.e., $t_{visibility}(c_k) \leq t_{visibility}(c_{k+1})$.*

Proof **Part (1).** This is implied by Part (1) of Claim 15 and the definition of $T_{j,notYetFIN}$.
Part (2). According to Algorithm 3, the server at $p_j \in \mathcal{P}$ does not remove the records in $\{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\} \cup \{maxT_{j,FINALIZED}\}$. The max function properties imply this part.

Claim 17 *Let $c_1, c_2 \in R_{legalExecution}$ be two system states that appear between $c_{j,in} \in R_{legalExecution}$ and $c_{j,out} \in R_{legalExecution}$, where $p_j \in \mathcal{P}$. It holds that $|(S_1 \cup S_2) \backslash (S_1 \cap S_2)| \leq \delta$, where $S_x \in \{1, 2\} = S_{p_i}$ in c_x.*

Proof By this lemma assumption, any read operation π in $R_{legalExecution}$ has at most δ concurrent write operations (Definition 15). Recall that $R_{legalExecution}$ does not include incomplete operations (Claim 5). Therefore, an update or an addition of the record $(t, \bullet)$ to S_{p_i} (between $c_{i,in}$ and $c_{i,out}$) implies that there is write operation $\pi_{w'}$ that is concurrent (Definition 14) with the read operation π. Thus, this claim. (Note that the same holds for this lemma's read operation, π_r.) □

Claim 18 *Let $t_{i,FINALIZED,c'} = t_{i,FINALIZED}$ denote the value of the sequence $t_{i,FINALIZED}$ in $c' \in R$. Let $c \in R_{legalExecution}$ be a system state that is between $c_{i,in}$ and $c_{i,out}$. The sequence $t_{i,FINALIZED,c}$ includes at most δ tags that are greater than $T(\pi_r)$, which are not in $t_{i,FINALIZED,c_{start}(\pi_r)}$.*

Proof Claim 14 implies that $T(\pi_r)$ is greater than the value of any element in $t_{i,FINALIZED,c_{start}(\pi_r)}$. From Claim 17, we get that, between $c_{start}(\pi_r)$ and $c_{i,out}$, Algorithm 3 may add to the sequence $t_{i,FINALIZED}$ at most δ records. Hence, the claim. □

Claim 19 *Let $c \in R_{legalExecution}$ be a system state that is between $c_{i,in}$ and $c_{i,out}$. It holds that $T(\pi_r) \in T_{i,readFinalized}$ (Definition 17) in c.*

Proof **Suppose that $\mathbf{c_{i,in}}$ appears before $\mathbf{c_{start}(ß_r)}$ in $\mathbf{R_{legalExecution}}$.**

We show that the conditions of Claim 15 hold in c and thus $T(\pi_r) \in T_{i,readFinalized}$. Specifically, we show that $T(\pi_r) \geq t_{visibility}(c)$ in c and that $(T(\pi_r), \bullet) \in S_{p_i}$ in c, because then we can complete the proof by using $T_{i,readFinalized} = T_{i,notYetFIN} \cup T_{i,FINALIZED}$ (Definition 17).

Let us look at the case in which c appears between $c_{i,in}$ and $\mathbf{c_{start}(ß_r)}$ in $\mathbf{R_{legalExecution}}$ (including both system states $\mathbf{c_{i,in}}$ and $\mathbf{c_{start}(ß_r)}$ as possible values of c).

We recall the fact that $T(\pi_r) \geq t_{visibility}(c_{start}(\pi_r))$ (Claim 13) and that $t_{visibility}(c_{start}(\pi_r)) \geq t_{visibility}(c)$ (Part (2) of Claim 16 and this case assumption that c appears no later than $c_{start}(\pi_r)$ in R). Thus, $T(\pi_r) \geq t_{visibility}(c)$.

To the end of showing that $(T(\pi_r), \bullet) \in S_{p_i}$ in c, we start by assuming that $c = c_{i,in}$ and then consider every system state c that appears between $c_{i,in}$ and $c_{start}(\pi_r)$ (including the latter state). Recall that $c_{i,in}$ is $R_{legalExecution}$'s first system state for which $(T(\pi_r), \bullet) \in S_{p_i}$ holds (cf. this lemma's statement). Therefore, $(T(\pi_r), \bullet) \in S_{p_j}$ implies $T(\pi_r) \in \{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\} \cup \{maxT_{j,FINALIZED}\} \subseteq T_{i,readFinalized}$ in $c = c_{i,in}$ (Part (2) of Claim 15).

Now, let us continue by assuming that c is the state in $R_{legalExecution}$ that immediately follows $c_{i,in}$ (and yet c does not appear in $R_{legalExecution}$ after $c_{start}(\pi_r)$). By the same arguments as above, it holds that $T(\pi_r) \geq t_{visibility}(c)$. Algorithm 3 does not include a line in which a server removes a record from its storage. Thus, we only need to show that $T(\pi_r)$ does not leave the set $T_{i,readFinalized}$ in the transition from $c_{i,in}$ to c. We show more than that, i.e., $T(\pi_r)$ does not leave the set $\{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\} \cup \{maxT_{j,FINALIZED}\} \subseteq T_{i,readFinalized}$ in the transition from $c_{i,in}$ to c.

We note that it cannot be the case that in the system state $c_{i,in}$ we have $T(\pi_r) \in \{maxT_{j,FINALIZED}\}$ and $T(\pi_r) \notin \{maxT_{j,FINALIZED}\}$ in c. The reason is that $T(\pi_r) \in \{maxT_{j,FINALIZED}\}$ in $c_{i,in}$ says that $T(\pi_r)$ is (either explicitly or implicitly) FINALIZED in $c_{i,in}$ and that status field in the record cannot change to a status that is not (either explicitly or implicitly) FINALIZED (Algorithm 3 and the way that Definition 17 constructs $t_{i,FINALIZED}$).

Suppose that in $c_{i,in}$ it holds that $T(\pi_r) \in \{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\}$ and in c it holds that $T(\pi_r) \notin \{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\}$. This implies that tag $T(\pi_r)$ becomes (either explicitly or implicitly) FINALIZED during that transition (Definition 17), That is, $T(\pi_r) \in \{maxT_{j,FINALIZED}\}$ in c and the proof is done.

The rest of the proof of this part is followed by repeating the same arguments for every two consecutive system states c' and c'' that are between $c_{i,in}$ and $c_{start}(\pi_r)$.

Let us look at the case in which c appears between $\mathbf{c_{start}(ß_r)}$ and $\mathbf{c_{i,out}}$ in $\mathbf{R_{legalExecution}}$ (including both system states $\mathbf{c_{start}(ß_r)}$ and $\mathbf{c_{i,out}}$ as possible values of c).

From the proof of the previous case, when $c = c_{start}(\pi_r)$, it holds that $T(\pi_r) \in \{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\} \cup \{maxT_{j,FINALIZED}\} \subseteq T_{i,readFinalized}$.

Recall also from the previous case that if Algorithm 3 causes $T(\pi_r)$ to leave the set $\{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\}$, then $T(\pi_r)$ becomes a member of the sequence $t_{i,FINALIZED}$ (Algorithm 3 and the way that Definition 17 constructs $t_{i,FINALIZED}$). From Claim 18, we get that Algorithm 3 may move $T(\pi_r)$ down the sequence $t_{i,FINALIZED}$, by including other (either explicitly or implicitly) FINALIZED records with higher tags, at most δ times between $c_{start}(\pi_r)$ and $c_{i,out}$ but still include $T(\pi_r)$ in $T_{i,FINALIZED}$. This implies $T(\pi_r) \in T_{i,readFinalized}$ (Definition 17) for the case in which c appears between $c_{start}(\beta_r)$ and $c_{i,out}$ in $R_{legalExecution}$, as well as the case in which $c_{i,in}$ appears before $c_{start}(\pi_r)$ in $R_{legalExecution}$.

Suppose that $\mathbf{c_{i,in}}$ appears after $\mathbf{c_{start}(\beta_r)}$ in $\mathbf{R_{legalExecution}}$.

By this case assumption, it holds that the tag $T(\pi_r)$ does not appear in the sequence $t_{i,FINALIZED}$ in $c_{start}(\pi_r)$. From Claim 18, we get that Algorithm 3 may include in the sequence $t_{i,FINALIZED}$ at most δ (either explicitly or implicitly) FINALIZED records with higher tags than $T(\pi_r)$ during the period that is between $c_{start}(\pi_r)$ and $c_{i,out}$. During this period, the record $(T(\pi_r), \bullet) \in S_{p_i}$ does appear in the storage of the server at p_i. By the arguments above, it appears either in $\{t \in T_{j,notYetFIN} : t \geq maxT_{j,FINALIZED}\}$ or in the top $\delta+1$ tags of $t_{i,FINALIZED}$. Therefore, $T(\pi_r) \in T_{i,FINALIZED}$ (Definition 17) and we can complete the proof by using $T_{i,readFinalized} = T_{i,notYetFIN} \cup T_{i,FINALIZED}$ (Definition 17).

□

Theorem 5 uses Lemma 11.

Theorem 5 (Only $\mathbf{T_{i,writeQuery}}$, $\mathbf{T_{i,readQuery}}$ and $\mathbf{T_{i,readFinalized}}$ are relevant and they are bounded) *Let $r = (t, \bullet) \in S_{p_i} : t \in \mathcal{T}$ be a record that the server at node $p_i \in \mathcal{P}$ stores in system state $c \in R_{legalExecution}$. Suppose that tag t is of relevance to c with respect to a (1) writer query request, (2) reader query request or (3) reader finalized request. The server at p_i stores the record $r = (t, w_j, \bullet) \in S_{p_j \in Q}$ and $r \in relevant(S_i)$, such that (1) $r \in T_{i,writeQuery}$, (2) $r \in T_{i,readQuery}$, and respectively, (3) $r \in T_{i,readFinalized}$ in c. Moreover, $|relevant(S_i)| \leq N + \delta + 3$, where $relevant(S_i) := T_{i,writeQuery} \cup T_{i,readQuery} \cup T_{i,readFinalized}$.*

Proof **Showing that $\mathbf{r \in T_{i,writeQuery}}$.**

The theorem assumption and Definition 16 imply that $T_{i,writeQuery} = \{\max\{t : (t, \bullet) \in S_{p_i}\}\}$ in c (Definition 17). The server at node p_i replies to a reader by returning the maximal tag t in any record stored in S_{p_i} (line 74). Therefore, the server at p_i and $T_{i,writeQuery}$ store in system state c any record $(t', \bullet)$ that is relevant with respect to a writer query request.

Showing that $\mathbf{r \in T_{i,readQuery}}$.

By this theorem assumption and Definition 16 it implies that $T_{i,readQuery} = \{\max\{t : (t, \bullet, d) \in S_{p_i} : d \in (\mathcal{D} \setminus \{\text{'pre'}\})\}\}$ in c (Definition 17). The server at node p_i replies to a reader by returning the tag t in any finalized or FINALIZED record stored in S_{p_i} (line 74). Therefore, the server at p_i and $T_{i,readQuery}$ store in system state c any record $(t', \bullet)$ that is relevant with respect to a reader query request.

Showing that $\mathbf{r \in T_{i,readFinalized}}$.

The proof of this case is implied by Lemma 11.

The bound $\mathbf{|relevant(S_i)| \leq N + + 3}$. This bound comes from the Definition 17, which implies $|T_{i,writeQuery}| \leq 1$ and $|T_{i,readQuery}| \leq 1$, as well as the definition of

1. Once the server at $p_i \in \mathcal{P}$ stores in S_{p_i} a record with a tag that is at least $t_{top} = (MAXINT, minID)$, where $minID := \max\{k : p_k \in \mathcal{P}\}$, the server at p_i suspends all responses to new write operations (at their query phase) while allowing the completion of the existing ones (until all servers agree on the highest finalized tag, cf. item 2), where $MAXINT \in \mathbb{Z}^+$, say, $MAXINT = 2^{64} - 1$. To that end, the server program tests whether $(maxPhase(\mathcal{D}) \leq t_{top})$ and respond to the writer only when the tests pass. That is, we modify line 75 to "else if $(maxPhase(\mathcal{D}) \leq t_{top})$ then reply$(j, (maxPhase(\mathcal{D}), \perp, \text{'qry'}))$."
2. While the invocation of new write operations is suspended by the modified line 75 (item 1), the gossip procedure keeps on propagating the maximal tags, which is $tagTuple()$'s returned value in lines 82 and 89 (as we show in Claim 20). Eventually, the servers at all nodes share the same triple of maximal tags. At that point in time, this Algorithm 3's variation uses the global reset procedure $globalReset(t)$ (Section 5.2) that (i) removes any record for all the server storages other than the ones with the tag $t = maxPhase(\mathcal{D} \setminus \{\text{'pre'}\})$, (ii) replaces the tag $t = (z, k)$ in that record with the tag $(1, k)$ and (iii) stops forever all ongoing client operations. To that end, between lines 88 and 89, the server program also includes the following if-statement: "if $(maxPhase(\mathcal{D}) \geq t_{top}) \wedge (\forall p_k \in \mathcal{P} : gossip[k] = tagTuple()) \wedge (tagTuple() = (t, t', t') \wedge t \geq t')$ then $globalReset(maxPhase(\mathcal{D} \setminus \{\text{'pre'}\}))$ else $S \leftarrow relevant(S)$," where $relevant(S)$ is taken from Theorem 5.

Fig. 3 A bounded extension of Algorithm 3

$T_{i,readFinalized}$ and Claim 17, which implies that during executions that have at most δ concurrent write operations, it holds that $|T_{i,readFinalized}| \leq N + \delta + 1$. □

Corollary 4 is implied directly from the definition of the set $relevant(S_i)$ (Theorem 5), Theorem 5 and line 82 to 89 of Algorithm 3.

Corollary 4 *Let $p_i \in \mathcal{P}$ be a node that hosts a server. The set $relevant(S_i)$ (Theorem 5) always includes the records $(t_{write}, \bullet)$, $(t_{read}, \bullet)$ and $(t_{anchor}, \bullet)$, which p_i gossips their tags in the triple $(t_{write}, t_{read}, t_{anchor})$ (line 89).*

10 A Bounded Variation on Algorithm 3

We present a variation of Algorithm 3 that has bounded message and state size. Figure 3 adds a couple of lines to the code of Algorithm 3 and uses the external building block *globalReset*() (Sect. 5.2). Theorem 13 demonstrates the correctness of the proposed variation. Note that the proof assumes the execution to be fair eventually in the manner of self-stabilizing systems in the presence of seldom fairness (Sect. 2.2). Namely, once the storage of at least one server includes at least one record with a tag t that is at least t_{top} (Fig. 3), we require the system execution to eventually be fair until all nodes return from the call to *globalReset*(). This requirement is indeed seldom, because such fair executions are needed only once in every $\mathcal{O}(z_{\max})$ write operations and during the recovery from rare transient faults (Theorem 3). After the recovery period and during the periods in which no server stores tag $t \geq t_{top}$, the execution is not required to be fair.

Definition 18 (*Legitimate overflows*) Le c be a system state in which every tag $t < t_{top}$ is smaller than the one that would trigger an overflow event. In this case, we say that c is overflow-free. We say that execution R has a legitimate overflow event if R's starting system state c is (i) both overflow-free and reset-free (Sect. 5.2.2), as well as (ii) the first step that immediately follows c includes the start (the first sent request to the server) of a pre-write phase that has the tag $t \geq t_{top}$. Let R'' be a suffix of $R = R' \circ R''$ that (a) includes a starting system state in which any $p_i \in \mathcal{P}$ (that hosts a server) stores a record $(t, \bullet) \in S_{p_i}$ with tag $t \geq t_{top}$ and (b) R' is the shortest matching prefix of R'' in R. In this case, we say that R'' is an execution with a legitimate overflow record. (Note that R'' may have system states, including the starting one, with tags $t' \geq t_{top}$, such that $t \neq t'$.)

Lemma 12 (Eventual recovery of Algorithm 3's variation in Fig. 3) *Let R be a fair execution of the bounded variation of Algorithm* 3 (Fig. 3). Suppose that in R's starting system state, c, it holds that there is a node $p_i \in \mathcal{P}$ (that hosts a server) stores a record $(t, \bullet) \in S_{p_i}$ with tag $t \geq t_{top}$ (but R is not necessarily an execution with a legitimate overflow record). Within $\mathcal{O}(\Psi)$ asynchronous cycles, R reaches a system state that is reset- and overflow-free.

Proof Recall that the reset procedure has a termination period within Ψ asynchronous cycles (Sect. 5.2.2). Thus, within Ψ asynchronous cycles, the system reaches a state $c' \in R$ that is reset-free. Note that if c' is also overflow-free, the proof is done. Therefore, we consider the complementary case and assume that c is reset-free but not necessarily overflow-free, i.e., $(t, \bullet) \in S_{p_i}$ with tag $t \geq t_{top}$. Claim 20 shows that within $\mathcal{O}(1)$ asynchronous cycles, the overflow handling proceeds to the invocation of the reset procedure (item 2 of Fig. 3), which in turn brings the system to a reset- and overflow-free state within Ψ asynchronous cycles. Therefore, the proof is done, because we showed that within $\mathcal{O}(\Psi)$ asynchronous cycles, the system reaches a state in R that is both reset- and overflow-free.

Claim 20 *Within $\mathcal{O}(1)$ asynchronous cycles, the system reaches a state, c', in which Condition* (1) *holds, where $t, t', t'' \in \mathcal{T} : t \leq t' \wedge t' \geq t''$.*

$$\begin{aligned} \exists p_i \in \mathcal{P} : (maxPhase_i(\mathcal{D}) = t' \geq t_{top}) \ \wedge \\ (\forall p_k \in \mathcal{P} : gossip_i[k] = tagTuple_i()) \ \wedge \\ (tagTuple_i() = (t', t'', t'')) \end{aligned} \tag{1}$$

Proof Suppose that this claim is false, and R includes a prefix R' with more than $\mathcal{O}(1)$ asynchronous cycles in which Condition (1) does not hold in every $c'' \in R'$.

We show that $\exists \mathbf{c_{stop}} \in \mathbf{R'} : \forall \mathbf{p_j} \in \mathcal{P} : (\mathbf{t}, \bullet) \in \mathbf{S_{p_j}} : \mathbf{t} \geq \mathbf{t_{top}}$.

Note that, within $\mathcal{O}(1)$ asynchronous cycles, the gossip protocol works correctly (Part (3) of Theorem 1). Moreover, the function $tagTuple()$ returns $(maxPhase(\mathcal{D})$, $maxPhase(\mathcal{D} \setminus \{\text{'pre'}\})$, $maxPhase(\{\text{'FIN'}\}))$ (line 92) and this triple is sent by the gossip service. This claim assumes that $(t, \bullet) \in S_{p_i}$, which implies that within $\mathcal{O}(1)$ asynchronous cycles of R', the system reaches a system state $c_{stop} \in R'$ for which $\forall p_j \in \mathcal{P} : (t, \bullet) \in S_{p_j} : t \geq t_{top}$ holds (Lemma 7).

We show that no step follows immediately after $\mathbf{c_{stop}}$ in which any server responds to a query request of a write operation.

No server responds to a query request due to the fact that $\forall p_j \in \mathcal{P} : (t, \bullet) \in S_{p_j}$ in c_{stop} and item 1 of Fig. 3.

We show that Condition (1) holds in $\mathbf{c}'' \in \mathbf{R}'$.

Every write operation that has started before c_{stop} terminates eventually (Theorem 4 with respect to Part (1) of Definition 10) or they cannot proceed beyond the pre-write phase. (This is because R is a fair execution and each write operation occurs within a constant number of phases and gossip rounds, we note that termination occurs within $\mathcal{O}(1)$ asynchronous cycles, because each phase occurs within $\mathcal{O}(1)$ asynchronous cycles, as we show in Part (3) of Theorem 1.) Let c''' be the first system state in which all of these write operations have terminated (or have stopped forever to proceed beyond the pre-write phase). Let $t' = \max_{p_j \in \mathcal{P}} maxPhase_j(\mathcal{D}) : t' \geq t \geq t_{top}$ and $t'' = \max_{p_k \in \mathcal{P}} maxPhase_k(\mathcal{D} \setminus \{\text{'pre'}\})$ in c'''. Recall that the server at p_j gossips $(t', \bullet)$ and the server at p_k gossips $(\bullet, t'', \bullet)$. Lemma 7 implies that within $\mathcal{O}(1)$ asynchronous cycles in R', the system reaches a state $c'''' \in R'$ in which $\forall p_\ell \in \mathcal{P} : tagTuple_\ell() = (t', t'', \bullet)$. By line 87, we have that Condition (1) holds in c'''' and so does this claim, because we have reached a contradiction with the assumption at the beginning of this proof.

□

Lemma 13 *Let R be a fair execution of the bounded variation of Algorithm 3* (Fig. 3) with a legitimate overflow record. (1) Within $\mathcal{O}(1)$ asynchronous cycles, the system reaches the first system state $c \in R$ in which it holds that there is a node $p_i \in \mathcal{P}$ (that hosts a server that) stores a record $(t, \bullet) \in S_{p_i}$ with tag $t \geq t_{top}$. Also, we can write $R = R' \circ R_{sameTagTuple} \circ R''$, such that (2) within a prefix R' of $\mathcal{O}(1)$ asynchronous cycles, the system reaches an unbounded suffix, $R_{sameTagTuple} \circ R''$, that has a prefix $R_{sameTagTuple}$ of $\mathcal{O}(1)$ asynchronous cycles, such that Condition (1) holds in its starting system state, $c' \in R_{sameTagTuple}$. Moreover, (3) only then at least one node calls $globalReset(t'')$, all nodes participate in that procedure and within $\mathcal{O}(\Psi)$ asynchronous cycles they resume, which leads to the end of $R_{sameTagTuple}$. Furthermore, (4) suppose that in $c' \in R_{sameTagTuple}$ it holds that $\exists p_j \in \mathcal{P} : \{(t'' = (z, k), \bullet, d) : d \in \mathcal{D} \setminus \{\text{'pre'}\}\} \subseteq S_{p_j}$, where t'' is the tag value taken from Condition (1). Then, there is system state $c'' \in R_{sameTagTuple}$ that follows c'' and in which $\forall p_\ell \in \mathcal{P} : S_{p_\ell} = \{((1, k), \bullet, \text{'FIN'})\}$. Otherwise, (5) $t'' = t_0$ (line 71) and $\forall p_\ell \in \mathcal{P} : S_{p_\ell} = \emptyset$ in c' and c''.

Proof **Part (1).** Lemma 5 implies this part of the proof.
Part (2). Claim 20 implies this part.
Part (3). Note that, by this lemma's assumption that R has a legitimate overflow record, the starting system state of R includes no tag that is greater or equal to t_{top}. Moreover, there is a node $p_i \in \mathcal{P}$ for which it holds that $\forall p_k \in \mathcal{P} : gossip_i[k] = (t', t'', t'')$ in c', where $t' \geq t_{top} \wedge t' \geq t''$, because of Part (2) of this proof which implies that Condition (1) holds in c'. Therefore, the only way in which $gossip_i[k] = (t', t'', t'')$ can hold in c', is if $tagTuple_k() = (t', t'', t'')$ holds in some system state that appears in R before (and perhaps also after) c' (and then these tags are gossiped from p_k to p_i), because these tag values do not appear in R's starting state. Moreover, at least one node calls $globalReset(t'')$ (due to item 2 of Fig. 3 and the fact that Condition (1) holds in c'). Therefore, all nodes resume within $\mathcal{O}(\Psi)$ asynchronous cycles (Sect. 5.2.2), which leads to the end of $R_{sameTagTuple}$.

Part (4). Claim 21 considers the case in which more than one node calls *globalReset*(t'') (item 2 of Fig. 3). This implies that all such calls during $R_{sameTagTuple}$ refer to the same FINALIZED tag $t'' = (\bullet, k)$. This part of the proof is implied by the fact that a call to *globalReset*(t'') indeed replaces t'' by $(1, k)$, cf. item 2 of Fig. 3.

Claim 21 *Suppose that $R_{sameTagTuple}$ includes two steps, a_i and a_j, in which p_i and p_j call globalReset$_i(t_i)$, and respectively, globalReset$_j(t_j)$. It is true that $t_i = t_j$.*

Proof We prove this claim by assuming that $t_i \neq t_j$ and then demonstrating a contradiction. Suppose, without the loss of generality, that a_i appears in $R_{sameTagTuple}$ before a_j. Let us start the proof by assuming that $t_i < t_j$ before considering the complementary case of $t_i > t_j$. We show that neither case is possible and, thus, this claim is correct.

The case of $\mathbf{t_i < t_j}$. By Part (2) of the proof of this lemma, we know that p_i calls *globalReset*$_i(t_i)$ in step a_i only after p_j has seen that the tag t_i is a FINALIZED record, because Condition (1) must hold with respect to t_i (item 2 of Fig. 3) and the definition of $tagTuple()$ (line 92). This is true starting from some system state that appears in R before the steps a_i and a_j.

When p_i takes step a_i and calls *globalReset*$_i(t_i)$, indeed, p_i has not seen t_j in a finalized (or FINALIZED) record, due to this case assumption that $t_i < t_j$ and the fact that Condition (1) holds with respect to tag t_i in the system state that immediately precedes a_i (item 2 of Fig. 3).

Once p_i takes step a_i and calls *globalReset*$_i(t_i)$, the function *globalReset*() disables p_i's server and therefore p_i's server does not receive or send in $R_{sameTagTuple}$ gossip messages after a_i. Therefore, p_i does not receive the tag t_j (in a finalized or FINALIZED record) in any step that follows a_i in $R_{sameTagTuple}$. Moreover, the fact that p_i does not gossip after a_i implies that p_j cannot receive from p_i a gossip message with $(\bullet, t_j, t_j)$ (lines 82 and 89). Thus, $gossip_j[i] = (\bullet, t_j, t_j)$ does not hold in any system state in $R_{sameTagTuple}$ that follows a_i. This is in contradiction to the assumption that p_j takes step $a_j \in R_{sameTagTuple}$, because this step requires Condition (1) to hold with respect to t_j (item 2 of Fig. 3).

The case of $\mathbf{t_i > t_j}$. By Part (2) of the proof of this lemma, we know that p_i calls *globalReset*$_i(t_i)$ in step a_i only after p_j has seen the tag t_i is a FINALIZED record, because Condition (1) must hold with respect to t_i (item 2 of Fig. 3) and the definition of $tagTuple()$ (line 92). This is true starting from some system state that appears in R before the steps a_i and a_j. The fact that, in the system state that immediately precedes step a_j in which p_j calls *globalReset*$_j(t_j)$, node p_j has indeed seen t_i in a finalized (or FINALIZED) record, demonstrates a contradiction due to this case assumption that $t_i > t_j$ and our assumption that a_j appears in R after a_i, because p_j should select t_j according to Condition (1) (item 2 of Fig. 3). □

Part (5). This part refers to a case in which no server stores any finalized record. Thus, by the arguments above, there is at least one node that calls *globalReset*$_j(t_0)$. Since no server stores any record, the tag t_0 is used (line 71), and this part of the proof flows simply from item 2 of Fig. 3. □

Theorem 6 (Bounded self-stabilizing $CAS(k_{threshold})$ in the presence of seldom fairness) *Algorithm* 3's variation in Fig. 3 is a bounded (message and state) size self-stabilizing algorithm (in the presence of seldom fairness) for implementing $T_{\text{CAS}(k_{threshold})}$'s task. Both the recovery and the overflow periods end within $\mathcal{O}(\Psi)$ asynchronous cycles.

Proof We demonstrate that the proposed algorithm is self-stabilizing (in the presence of seldom fairness). To that end, we show that (1) the proposed algorithm can always recover within $\mathcal{O}(\Psi)$ asynchronous cycles from an arbitrary starting system state of a fair execution, (2) during arbitrary executions that start from a legitimate system state, the system execution is legal, but (3) once in every $\mathcal{O}(z_{\max})$ write operations, the system stops providing liveness until the system execution becomes fair and then within $\mathcal{O}(\Psi)$ asynchronous cycles (during which safety is not violated) liveness is regained.
Part (1). Theorem 4 (with respect to Part (1) of Definition 10) demonstrates that the operations of Algorithms 3 always terminate. Lemma 12 demonstrates that the added mechanisms for dealing with overflow events (Fig. 3) always finish to deal with overflows and then the system simply follows Algorithms 3 for a period of at least $z_{\max}$ write operations.

The proof of Theorem 2 considers a complete write operation, π_{write}, that its tag is greater than any tag that is present throughout any earlier stage of the recovery process, i.e., including the set of tags that appeared in the starting system state. As long as no overflow event occurs, within $\mathcal{O}(1)$ asynchronous cycles, the system can complete the write operation, π_{write}, such that $T(\pi_{write})$'s record stays at the set $relevant(S_i)$ at least until a later write operation is completed. If an overflow handling is needed, recovery occurs within $\mathcal{O}(\Psi)$ asynchronous cycles (Lemma 13).
Part (2). Algorithm 3's correctness (Theorems 3 and 4) implies this part.
Part (3). Lemma 13 implies this part. □

11 Cost Analysis

The main complexity measures of self-stabilizing systems in the presence of seldom fairness are (Sect. 2.2.6): (i) the maximum length overflow period, which is of $\mathcal{O}(\Psi)$ asynchronous cycles (Theorem 6) for the proposed solution, and (ii) the maximum length of the period during which the system recovers after the occurrence of transient failures, which is $\mathcal{O}(1)$ asynchronous cycles for the case of the unbounded solution (Theorem 2) but can also take $\mathcal{O}(\Psi)$ asynchronous cycles if the recovery period includes an overflow (or a recovery of the overflow mechanism).

Cadambe et al. [15] present a version of the $CAS(k_{threshold})$ algorithm that includes elements of garbage collection that recycles merely the stored objects and never the meta-data, i.e., it never removes the records themselves, because the garbage collector removes only the coded elements and always keeps the tags and the phase indices. In the context of self-stabilizing systems, this implies that the storage is unbounded, because a single transient fault can clog the storage. Cadambe et al. also explicitly say that when the execution is unfair, an infinite storage is required [15, Table 1]. One of the advantages of Algorithm 3's variation in Fig. 3 is that it offers bounded local storage of $\mathcal{O}(N+\delta)$ records also during periods in which the execution is unfair (Theorem 5), i.e., $\mathcal{O}((\log_2 |\mathcal{V}|)(N+\delta))$ bits in total.

The proposed solution has write operations that include four phases rather than three, as in [15]. Cadambe et al. [15, Theorem 4] analyze the communication costs of $CAS(k_{threshold})$ and show that they can be made as small as $\frac{N}{N-f}\log_2 |\mathcal{V}|$ bits by choosing $k = N - 2f$. Moreover, Algorithm 3 and its variation in Fig. 3 consider gossip messages that include three tags, rather than just one as in Cadambe et al. [15]. When comparing the communication costs of a self-stabilizing algorithm to another that does not consider recovery from transient faults, we have to take into consideration that fact that self-stabilizing algorithms can never stop communicating (because then the system can be first brought to a state in which communication stops, and then a transient-fault merely change some part of that state, which the algorithm cannot correct because it has stopped communicating, see [18, Chapter 3.2] for details). Therefore, the proposed algorithm never stops sending gossip messages, whereas the one by Cadambe et al. [15, Theorem 4] sends $\mathcal{O}(N^2)$ gossip messages per client operation.

12 Discussion

We studied the implementation of a private coded atomic storage protocol, which is resilient to data-corrupting malicious servers. For the case of asynchronous message-passing networks that provide fair communication, we proposed a self-stabilizing algorithm that preserves privacy and recovers after the occurrence of transient faults. Our solution requires the system to first reach a fair execution before the algorithm guarantees recovery. Moreover, once in a practically infinite number of write operations, the proposed solution again requires fair execution to the end of dealing with counter overflows. Since overflow events of 64-bit integers, in any practical setting, can only be the result of transient faults, and since transient faults are very rare, we believe that our novel stabilization criteria are applicable to a range of similar problems that require self-stabilizing tag schemes. Thus, as future work, we propose to study self-stabilizing (in the presence of seldom fairness) of consensus [9, 21], virtual synchrony [29], and other shared register emulation schemes [16], to name a few.

12.1 Extension: Recyclable Client Identifiers

In Sect. 2.2.3, we assume that clients that crash, do not return to take steps. We present here an elegant extension that is based on well-known techniques. This extension allows the nodes to recycle their client identifiers whenever they resume operation after failing. That is, we tolerate crash failures of the client nodes (rather than crashes) that are followed by detectable restarts. For dealing with detectable restarts of the server nodes, we point out the existence of self-stabilizing quorum reconfiguration [27, 28, 28].

The client identifier could be a pair that includes the identifier of node p_i and an incarnation number that is incremented whenever a failing node resumes and then wishes to invoke client operations. A well-known technique for maintaining a persistent incarnation number (without assuming access to stable storage or that the storage is not prone to corruption by transient-faults) is to let a quorum service emulate a shared counter, similar to the one in [25, Section 3.3]. Namely, p_i queries all servers for its current maximal incarnation number, and waits for a quorum of replies. Then, the client sends to all servers its updated incarnation number, which is the maximum in all query replies plus one, and waits for a quorum of replies before invoking the next client operation. Note that each node p_i that hosts a client has to maintain its own incarnation counter (by using the client identifier for partitioning the space of incarnation numbers) and that the above procedure is executed only when p_i resumes after failing.

We end the description of this extension by saying that the bounded variation of Algorithm 3 (Sect. 3) includes a global reset procedure that resets all clients and servers. This reset procedure can also reset the above mechanisms for recyclable client identifiers. In addition, whenever the incarnation number reaches its maximum value, the global reset procedure is triggered. Note that the latter happens only after the occurrence of a transient fault. Thus, the client never runs out of incarnation numbers (in any participial settings). The description of the above procedure would not be complete without saying that the server gossip periodically the set of all known pairs of client identifiers and their incarnation numbers. When such a set is received, the server updates its local set by adding all the pairs that come from clients that it does not know and updating all the existing pairs with the highest incarnation number. Note that the size of these sets is bounded by the number of possible clients, N.

12.2 Conclusions

We have presented the first self-stabilizing algorithm in the presence of seldom fairness. On the one hand, Dijkstra's self-stabilization criterion [17] requires a bounded recovery period but its well-known solutions usually model crashes as transient faults [18]. On the other hand, the less restrictive criteria of pseudo-self-stabilizing [12, 23] and practically-self-stabilizing systems [9, 21, 29], do not model crash failures as transient faults, but also do not offer a bounded recovery period. We view the criteria of self-stabilizing algorithms (in the presence of seldom fairness) as an attractive alternative to both Dijkstra's self-stabilization criterion [17], and the less restrictive criteria of pseudo-self-stabilizing [12, 23] and practically-self-stabilizing systems [9, 21, 29].

We consider self-stabilizing systems (in the presence of seldom fairness) to be (i) wait-free (since they do not assume execution fairness) in the absence of transient faults, and (ii) offering a bounded recovery period from transient faults, as in Dijkstra's self-stabilization criterion [17]. This is offered at the expense of compromising liveness (without jeopardizing safety) for a bounded period that occurs once in every practically infinite number of operations, of say, 2^{64}.

Acknowledgements We thank Robert Gustafsson and Andreas Lindhé for the useful discussions and for helping to improve the presentation significantly. We are grateful for the comments received from anonymous reviewers.

Funding Open access funding provided by Chalmers University of Technology. The work of E. M. Schiller was partially supported by the project 'Privacy-Protected Machine Learning for Transport Systems' of Area of Advance Transport and Chalmers AI Research Centre (CHAIR) as well as by AutoSPADA (Automotive Stream Processing and Distributed Analytics) OODIDA Phase 2 by Vinnova's FFI framework (reference number 2019-05884).

Declarations

Conflict of interest All authors certify that there is no actual or potential conflict of interest in relation to this article.

References

1. Alon, N., Attiya, H., Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Practically stabilizing SWMR atomic memory in message-passing systems. J. Comput. Syst. Sci. **81**(4), 692–701 (2015)
2. Androulaki, E., Cachin, C., Dobre, D., Vukolic, M.: Erasure-coded Byzantine storage with separate metadata. In: Aguilera, M.K., Querzoni, L., Shapiro, M. (eds.) Principles of Distributed Systems—18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16–19, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8878, pp. 76–90. Springer (2014)
3. Arora, A., Gouda, M.G.: Closure and convergence: a formulation of fault-tolerant computing. In: FTCS, pp. 396–403. IEEE Computer Society (1992)
4. Attiya, H.: Robust simulation of shared memory: 20 years after. Bull. EATCS **100**, 99–113 (2010)
5. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM (JACM) **42**(1), 124–142 (1995)
6. Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-stabilization by local checking and global reset (extended abstract). In: Tel, G., Vitányi, P.M.B. (eds.) Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29–October 1, 1994, Proceedings, Lecture Notes in Computer Science, vol. 857, pp. 326–339. Springer (1994)
7. Bao, Z., Wang, Q., Shi, W., Wang, L., Lei, H., Chen, B.: When blockchain meets SGX: an overview, challenges, and open issues. IEEE Access **8**, 170404–170420 (2020)
8. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: 20th Symp. on Theory of Computing, pp. 1–10. ACM (1988)
9. Blanchard, P., Dolev, S., Beauquier, J., Delaët, S.: Practically self-stabilizing Paxos replicated state-machine. In: Noubir, G., Raynal, M. (eds.) Networked Systems—Second International Conference, NETYS 2014, Marrakech, Morocco, May 15–17, 2014. Revised Selected Papers, Lecture Notes in Computer Science, vol. 8593, pp. 99–121. Springer (2014)
10. Bonomi, S., Dolev, S., Potop-Butucaru, M., Raynal, M.: Stabilizing server-based storage in Byzantine asynchronous message-passing systems: extended abstract. In: Georgiou, C., Spirakis, P.G. (eds.) Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21–23, 2015, pp. 471–479. ACM (2015). http://dl.acm.org/citation.cfm?id=2767386
11. Bonomi, S., Pozzo, A.D., Potop-Butucaru, M., Tixeuil, S.: Brief announcement: Optimal self-stabilizing mobile Byzantine-tolerant regular register with bounded timestamps. In: SSS, Lecture Notes in Computer Science, vol. 11201, pp. 398–403. Springer (2018)
12. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. Distrib. Comput. **7**(1), 35–42 (1993)

13. Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded Byzantine distributed storage. In: 2006 International Conference on Dependable Systems and Networks (DSN 2006), 25–28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings, pp. 115–124. IEEE Computer Society (2006). http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=10881
14. Cadambe, V.R., Wang, Z., Lynch, N.A.: Information-theoretic lower bounds on the storage cost of shared memory emulation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25–28, 2016, pp. 305–313 (2016)
15. Cadambe, V.R., Lynch, N.A., Médard, M., Musial, P.M.: A coded shared atomic memory algorithm for message passing architectures. Distrib. Comput. **30**(1), 49–73 (2017)
16. Cadambe, V.R., Nicolaou, N.C., Konwar, K.M., Prakash, N., Lynch, N.A., Médard, M.: ARES: adaptive, reconfigurable, erasure coded, atomic storage (2018). CoRR arXiv:1805.03727
17. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
18. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
19. Dolev, S., Herman, T.: Dijkstra's self-stabilizing algorithm in unsupportive environments. In: Datta, A.K., Herman, T. (eds.) Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1–2, 2001, Proceedings, Lecture Notes in Computer Science, vol. 2194, pp. 67–81. Springer (2001)
20. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Trans. Parallel Distrib. Syst. **8**(4), 424–440 (1997)
21. Dolev, S., Kat, R.I., Schiller, E.M.: When consensus meets self-stabilization. J. Comput. Syst. Sci. **76**(8), 884–900 (2010)
22. Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Stabilizing data-link over non-FIFO channels with optimal fault-resilience. Inf. Process. Lett. **111**(18), 912–920 (2011)
23. Dolev, S., Dubois, S., Potop-Butucaru, M.G., Tixeuil, S.: Crash resilient and pseudo-stabilizing atomic registers. In: Baldoni, R., Flocchini, P., Ravindran, B. (eds.) Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18–20, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7702, pp. 135–150. Springer (2012)
24. Dolev, S., Hanemann, A., Schiller, E.M., Sharma, S.: Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-FIFO) dynamic networks. In: Stabilization, Safety, and Security of Distributed Systems—14th Int. Sym., SSS 2012, LNCS, vol .7596, pp. 133–147. Springer (2012)
25. Dolev, S., Georgiou, C., Marcoullis, I., Schiller, E.M.: Practically stabilizing virtual synchrony (2015). CoRR arXiv:1502.05183
26. Dolev, S., Petig, T., Schiller, E.M.: Brief announcement: robust and private distributed shared atomic memory in message passing networks. In: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21–23, 2015, pp. 311–313 (2015)
27. Dolev, S., Georgiou, C., Marcoullis, I., Schiller, E.M.: Self-stabilizing reconfiguration. In: Middleware Posters and Demos, pp. 13–14. ACM (2016)
28. Dolev, S., Georgiou, C., Marcoullis, I., Schiller, E.M.: Self-stabilizing reconfiguration. In: NETYS, Lecture Notes in Computer Science, vol. 10299, pp. 51–68 (2017), the complementary technical report can be found at CoRR arXiv:1606.00195
29. Dolev, S., Georgiou, C., Marcoullis, I., Schiller, E.M.: Practically-self-stabilizing virtual synchrony. J. Comput. Syst. Sci. **96**, 50–73 (2018)
30. Dolev, S., Petig, T., Schiller, E.M.: Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks (2018). CoRR arXiv:1806.03498
31. Dutta, P., Guerraoui, R., Levy, R.R., Vukolic, M.: Fast access to distributed atomic memory. SIAM J. Comput. **39**(8), 3752–3783 (2010)
32. Duvignau, R., Raynal, M., Schiller, E.M.: Self-stabilizing byzantine-tolerant broadcast (2022). CoRR arXiv:2201.12880
33. Fan, R., Lynch, N.A.: Efficient replication of large data objects. In: Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1–3, 2003, Proceedings, LNCS, vol. 2848, pp. 75–91. Springer (2003)
34. Gemmell, P., Sudan, M.: Highly resilient correctors for polynomials. Inf. Process. Lett. **43**(4), 169–174 (1992)

35. Georgiou, C., Nicolaou, N.C., Shvartsman, A.A.: Fault-tolerant semifast implementations of atomic read/write registers. J. Parallel Distrib. Comput. **69**(1), 62–79 (2009)
36. Georgiou, C., Gustafsson, R., Lindhe, A., Schiller, E.M.: Self-stabilization overhead: an experimental case study on coded atomic storage (2018). CoRR arXiv:1807.07901
37. Georgiou, C., Gustafsson, R., Lindhé, A., Schiller, E.M.: Self-stabilization overhead: a case study on coded atomic storage. In: NETYS'19, Lecture Notes in Computer Science, vol. 11704, pp. 131–147. Springer (2019), the complemetry technical report appears in CoRR arXiv:1807.07901, 2018
38. Gilbert, S., Lynch, N.A., Shvartsman, A.A.: Rambo: a robust, reconfigurable atomic memory service for dynamic networks. Distrib. Comput. **23**(4), 225–272 (2010)
39. Gramoli, V., Nicolaou, N., Schwarzmann, A.A.: Consistent distributed storage. In: Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2021)
40. Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead Byzantine fault-tolerant storage. In: Bressoud, T.C., Kaashoek, M.F. (eds.) Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14–17, 2007, pp. 73–86. ACM (2007)
41. Johnen, C., Higham, L.: Fault-tolerant implementations of regular registers by safe registers with applications to networks. In: Garg, V.K., Wattenhofer, R., Kothapalli, K. (eds.) Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3–6, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5408, pp. 337–348. Springer (2009)
42. Konwar, K.M., Prakash, N., Kantor, E., Lynch, N.A., Médard, M., Schwarzmann, A.A.: Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23–27, 2016, pp. 720–729. IEEE Computer Society (2016). http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7510487
43. Konwar, K.M., Prakash, N., Lynch, N.A., Médard, M.: RADON: repairable atomic data object in networks. In: Fatourou, P., Jiménez, E., Pedone, F. (eds.) 20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13–16, 2016, Madrid, Spain, LIPIcs, vol. 70, pp. 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016). http://www.dagstuhl.de/dagpub/978-3-95977-031-6
44. Konwar, K.M., Prakash, N., Lynch, N.A., Médard, M.: A layered architecture for erasure-coded consistent distributed storage. In: Schiller, E.M., Schwarzmann, A.A. (eds.) Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25–27, 2017, pp. 63–72. ACM (2017)
45. Lamport, L.: On interprocess communication. Part II: algorithms. Distrib. Comput. **1**(2), 86–101 (1986)
46. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
47. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24–27, 1997, pp. 272–281. IEEE Computer Society (1997). http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4855
48. McEliece, R.J., Sarwate, D.V.: On sharing secrets and Reed-Solomon codes. Commun. ACM **24**(9), 583–584 (1981)
49. Musial, P.M., Nicolaou, N.C., Shvartsman, A.A.: Implementing distributed shared memory for dynamic networks. Commun. ACM **57**(6), 88–98 (2014)
50. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. J. Soc. Ind. Appl. Math. **8**(2), 300–304 (1960)
51. Roth, R.M.: Introduction to Coding Theory. Cambridge Press, Cambridge (2006)
52. Salem, I., Schiller, E.M.: Practically-self-stabilizing vector clocks in the absence of execution fairness. In: Networked Systems—6th International Conference, NETYS 2018, p to appear (2018)
53. Sampson, A., Nelson, J., Strauss, K., Ceze, L.: Approximate storage in solid-state memories. ACM Trans. Comput. Syst. **32**(3), 9:1-9:23 (2014)
54. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
55. Skeen, D.: A quorum-based commit protocol. In: Berkeley Workshop, pp. 69–80 (1982)
56. Spiegelman, A., Cassuto, Y., Chockler, G.V., Keidar, I.: Space bounds for reliable storage: fundamental limits of coding. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25–28, 2016, pp. 249–258 (2016)
57. Welch, L.R., Berlekamp, E.R.: Error correction for algebraic block codes. US Patent 4,633,470. https://www.google.com/patents/US4633470 (1986)

58. Xing, B.C., Shanahan, M., Leslie-Hurd, R.: Intel® software guard extensions (intel® SGX) software support for dynamic memory allocation inside an enclave. In: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016, pp. 11:1–11:9. ACM (2016)